RUNNING MS-DOS® QBasic™

Michael Halvorson
David Rygmyr

- Get started with the programming environment in MS-DOS versions 5 and 6
- Learn fundamental and advanced programming techniques
- Write modern, structured Basic programs

Microsoft PRESS®

The Authorized Edition

RUNNING MS-DOS® QBasic

RUNNING

MS-DOS® QBasic

Michael Halvorson
and David Rygmyr

- Get started with the programming environment in MS-DOS versions 5 and 6
- Learn fundamental and advanced programming techniques
- Write modern, structured Basic programs

Microsoft
PRESS
®

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 1991 by Microsoft Press, a division of Microsoft Corporation

All rights reserved. No part of the contents of this book
may be reproduced or transmitted in any form or by any means
without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Halvorson, Michael.
Running MS-DOS QBasic / Michael Halvorson, David Rygmyr.
p. cm.
Includes index.
ISBN 1-55615-340-6
1. QBasic (Computer program language) 2. MS-DOS (Computer
operating system) I. Rygmyr, David, 1958- . II. Title.
QA76.73.Q33H35 1991
005.26'2--dc20 90-25118
CIP

Printed and bound in the United States of America.

6 7 8 9 AGAG 4 3

Distributed to the book trade in Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182–190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Apple and Macintosh are registered trademarks of Apple Computer, Incorporated.
IBM is a registered trademark of International Business Machines Corporation.
Microsoft and MS-DOS are registered trademarks and Microsoft QuickBasic,
Visual Basic, and Windows are trademarks of Microsoft Corporation.

Project Editor: Erin O'Connor
Technical Editor: Dail Magee, Jr.

For all our hardworking friends at Microsoft Press.

Contents

Acknowledgments ix

Introduction xi

1

Introduction to Programming and QBasic 1

2

Getting Started with QBasic 9

3

Introduction to the QBasic Language 37

4

QBasic Variables and Operators 47

5

Controlling Program Flow 79

6

Working with QBasic Loops 117

7

Creating Your Own Subprograms and Functions 153

8

Working with Large Amounts of Data 185

9

Working with Strings 231

10

Working with Files 281

11

Working with Graphics and Sound 331

12

Debugging QBasic Programs 381

13

Learning More About Basic 403

Appendix A

Using QBasic Menus and Options 409

Appendix B

ASCII and IBM Extended Character Sets 423

Appendix C

QBasic Statements and Functions 427

Appendix D

Converting GW-BASIC Programs to Run Under QBasic 431

Appendix E

Solutions to Questions and Exercises 441

Index 471

Acknowledgments

Many skilled professionals at Microsoft Press worked hard to put this book into your hands. We gratefully acknowledge Darcie Furlan for her effective design; Becky Geisler-Johnson and Roger Collier for their superb illustrations; Debbie Kem and Judith Bloch for their speedy word processing and coding; Peggy Herman for her skillful coordination of the layout and pasteup; Shawn Peck and Deborah Long for their careful proofreading; and Carolyn Magruder for her adroit typesetting. Special thanks also to project editor Erin O'Connor for her high-performance editorial and management skills and to technical editor Dail Magee, Jr., for creating the screen images and for pursuing perfection relentlessly.

Michael Halvorson
David Rygmyr
February 1993

SPECIAL OFFER

Companion Disk for Running MS-DOS® QBasic™

Get a head start on learning how to program with MS-DOS QBasic. Microsoft Press has created a timesaving Companion Disk, available in 3.5-inch format and 5.25-inch format, that contains all the programs from *Running MS-DOS QBasic*. Now you don't have to spend hours typing in the more than 110 program listings from the book. This Companion Disk is guaranteed to save you time and effort!

Domestic Ordering Information:
To order, use the special reply card in the back of the book. If the card has already been used, please send $18.95, plus sales tax in the following states if applicable: AZ, CA, CO, CT, DC, FL, GA, HI, ID, IL, IN, IA, KS, KY, ME, MD, MA, MI, MN, MS, MO, NE, NV, NJ, NM, NY, NC, OH, OK, PA, RI, SC, TN, TX, UT, VA, VT, WA, WV, WI. Microsoft reserves the right to correct tax rates and/or collect the sales tax assessed by additional states as required by law, without notice. Please add $5.00 per disk set for domestic postage and handling charges. Mail your order to: **Microsoft Press, Attn: Companion Disk Offer, 21919 20th Ave. SE, Box 3011, Bothell, WA 98041-3011.** Specify 3.5-inch (item number 097-000-486) or 5.25-inch format (item number 097-000-483). Payment must be in U.S. funds. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, or MasterCard; please include credit card number, expiration date, and cardholder signature. Allow 2–3 weeks for delivery.

Foreign Ordering Information (within the U.K. and Canada, see below):
Follow procedures for domestic ordering. Add $15.00 per disk set for foreign postage and handling.

U.K. Ordering Information:
Send your order in writing along with £17.95 (includes VAT) to: Microsoft Press, 27 Wrights Lane, London W8 5TZ. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, MasterCard, or Diners Club; please include credit card number, expiration date, and cardholder signature. Specify 3.5-inch or 5.25-inch format.

Canadian Ordering Information:
Send your order in writing along with $26.95 (includes GST) to: Macmillan Canada, Attn: Microsoft Press Department, 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7. You may pay by check or money order (payable to Microsoft Press) or by VISA or MasterCard; please include credit card number, expiration date, and cardholder signature. Specify 3.5-inch or 5.25-inch format.

Microsoft Press Companion Disk Guarantee:
If a disk is defective, a replacement disk will be sent. Please send the defective disk and your packing slip (or copy) to: Microsoft Press, Consumer Sales, One Microsoft Way, Redmond, WA 98052-6399.

Introduction

IBM introduced the original IBM Personal Computer in August 1981. The base model PC, which listed for $1,565, contained an 8088 microprocessor, featured 16 KB of main memory, and required that you supply a cassette player for data storage and a television for display. That early system seems bare-bones by today's standards, but one celebrated tradition that originated with the first IBM PC remains intact: IBM PCs and compatibles are always sold with a version of MS-DOS and a version of the Basic programming language.

This book will help you take part in the continuing MS-DOS and Basic success story. It will help you join the millions of computer users who use Basic to manage their computers and increase their day-to-day productivity. Some use Basic to create MS-DOS applications for profit—skilled Basic programmers are in high demand—and others write Basic programs just for the fun of it.

The version of Basic we'll teach you to use is MS-DOS QBasic, the new, powerful Basic that comes free with MS-DOS versions 5 and 6. If you have MS-DOS 5 or 6 installed on your computer, you have everything you need to get started—you can write your first QBasic program today.

Is This Book for You?

We designed this book to be a thorough introduction to modern QBasic programming for the beginner with no previous programming experience. If you've wondered about programming but haven't known how to get started, this is the book for you. We introduce the important programming concepts one at a time—starting at the *beginning*—and reinforce the lessons throughout the book with practice sessions, questions, and programming exercises. In the back of the book, we provide solutions to the questions and exercises.

Programming can be hard work, but you'll find your efforts rewarding. In addition to learning the fundamentals of QBasic, you'll create useful and interesting MS-DOS applications—programs you can put to work for you day in and day out. Here are a few of the applications you'll build in this book:

- A program that calculates the price of an item, including sales tax.
- A golden-age television trivia program.
- A simple monthly budgeting program.
- A program that rolls electronic dice.
- A program that produces interesting sound effects.
- A program that displays the verses and chorus of the traditional American song "Clementine."
- A program that tracks monthly sales information for a beverage distributor.
- A music database program you can modify to track any type of database information.
- A program that simulates zooming through space.

Altogether, this book contains more than 130 complete programs that you can type in and run—plenty of hands-on practice to get you up and running.

Have You Used Basic Before?

If you've used an older version of Basic, such as GW-BASIC or BASICA, this book will introduce you to the benefits of the much-improved, modern QBasic language and show you how you can convert your existing Basic programs to run under QBasic. QBasic's powerful editing environment, on-line help facilities, and language enhancements will improve the quality of your existing programs and help you build new programs faster than ever before. If you're already comfortable with the fundamentals of Basic, we recommend that you skim Chapters 1 through 6 and start reading in earnest at Chapter 7. We also recommend that you read Appendix D, which summarizes the differences between QBasic and GW-BASIC and provides some tips for converting your GW-BASIC programs to QBasic.

How This Book Is Organized

If you're starting from scratch, we recommend that you take up the Basic programming topics in the sequence in which the book presents them.

Basic's ability to perform some fairly impressive tasks rests on simple fundamentals, so you won't want to miss any of the early details. Chapters 1 through 6 introduce the important first concepts of Basic programming—designing programs, using QBasic's programming environment, writing instructions, working with variables, setting up decision structures, and using loops.

Chapters 7 through 12 build on these fundamentals as they introduce you to more sophisticated practices—using subprograms and functions to organize your programs into modules, arrays to manage data, strings to process text, and files to store information. You'll learn how to make your programs produce printouts, how to use color and graphics to make things attractive, and how to use sound to impress your friends. Finally, you'll learn about debugging practices that will ease the pain of your occasional mistakes.

Running MS-DOS QBasic concludes with five appendixes. Appendix A summarizes the mouse and keyboard commands in the QBasic programming environment and covers the customization and startup options available to you. We recommend that you consult Appendix A if you haven't used a menu-driven application and need to review general operations such as selecting text and using a dialog box. Appendix B lists the ASCII and IBM extended character sets, which together document every character you can display using QBasic. Appendix C lists the complete set of QBasic statements and functions. Appendix D describes converting GW-BASIC programs to run under QBasic—recommended reading if you have previous experience with Basic. Appendix E gives complete solutions for all the questions and programming exercises in the book.

The Future of Basic

Learning to program in QBasic will require some patience, but rest assured that your investment will be of value for years to come. In addition to QBasic, Microsoft Corporation sells and supports three powerful Basic products for recreational and professional programmers: Microsoft QuickBasic for the Apple Macintosh, Microsoft Visual Basic for MS-DOS, and Microsoft Visual Basic for Windows. If your needs grow beyond QBasic, you have advanced versions of Basic to turn to.

Microsoft has added Basic-like macro languages to several of its applications for Windows, including Microsoft Word for Windows and

Microsoft Access. If you have either of these products, you can put your new Basic skills to work immediately.

Microsoft plans to use a subset of Basic as a unifying language within one or more of its operating environments. This means that you will be able to control the overall operation of your computer with Basic and make your applications work together as players on the same team, rather than distant rivals. Learning QBasic now will enable you to write useful programs today and prepare you for the future of computing.

CHAPTER 1

Introduction to Programming and QBasic

WHY DO PEOPLE USE COMPUTERS?

A computer is simply a tool. Its purpose is to serve you—to help you perform a task quickly and efficiently. You don't need to be convinced of that. The very fact that you bought this book means you've recognized that a computer can be of use to you. But you might wonder why you would want to write computer programs of your own.

Why Should *I* Learn to Program?

After all, today you can choose from thousands of ready-made software programs—programs that lead you through the maze of tax-return preparation as easily as they challenge you with the maze in an adventure game. Confronted by such a wide variety of commercial software, you might believe that the market is tapped out—that there's a program to suit your every need.

But you'd be wrong.

Programming as a problem solver

Stereotypes notwithstanding, professional programmers are also business people. For the most part, they design programs that fill the broadest needs of the market—it's just not economically feasible to write a program that only a few people will buy. Translation: The more specialized your needs, the less likely you are to find an appropriate ready-made software program.

By learning to program, you can custom-design programs to meet your specific needs.

Programming as a learning tool

Another reason for learning to program is that the process teaches you how computers work on the inside: You come to appreciate the step-by-step logic that results in the flashy displays and the slick printouts. In much the way that an architect can look at the exterior of a building and see the organization of the steel beams, pipes, and support structures underneath it, a person who learns the basics of computer programming begins to get a feel for what a computer program does behind the scenes. Such familiarity also gives you a feel for what a computer can and can't do. For this reason, an introductory course in computer programming is often the first class required in the study of computers in high school or college. Plus, by learning the groundwork, you're learning to speak the language of the "pros." Familiarity with programming terms and concepts helps you express yourself clearly.

Programming as recreation

The final reason for writing your own programs is that it can be downright *fun.* For beginners and experienced programmers alike, computer programming can be enjoyable and rewarding.

As you've seen, there are several good reasons for you to learn to program. Now let's take a moment to talk terminology. What *is* a computer program, anyway?

Custom-designed Programming

What if your needs are pretty straightforward? Perhaps you just need some organizational programs to simplify your life. QBasic lets you create programs that are as simple or as detailed as you need. Here are a few people who could benefit by writing their own QBasic programs:

- A baseball enthusiast who wants to record team and player statistics and instantly recall items of special interest
- A video-game devotee who wants to build a better adventure game
- A manager or small-business owner who wants to track appointments and important dates and be notified automatically about pending events
- A student or scientist who needs to perform specialized mathematical calculations
- A homeowner who wants to track household finances
- A doctor who wants to track patient records, prescriptions, and billing but wants to go beyond the limitations of traditional bookkeeping tools

WHAT IS A COMPUTER PROGRAM?

To perform useful work, a computer must consist of two things: *hardware* and *software.*

- Hardware is the physical parts of the computer that you can see and touch, such as the keyboard and the screen.
- Software is a bit more elusive: It's the intelligence that controls the hardware and allows you to actually use the computer.

You can't use one without the other: Hardware without software is like a record player without records.

A *computer program* is software. It's simply a set of instructions that collectively cause a computer to perform a useful task, such as calculation or word processing. A program can be as short as a single instruction, or it can contain hundreds or even thousands of individual instructions.

Computers: Heroes or Villains?

Movies and television have done a fine job of showing us how computers can take away jobs, make life more difficult, and take over the world. Not exactly the kind of tool you'd want to take into your home or business, much less use or program!

Despite this bad press, personal computers have entered the workplaces and homes of millions, allowing people to perform useful work that makes their lives easier and more productive. People are starting to change their minds about these strange electronic devices. What once churned out reams of inscrutable punch cards and often took up more than its share of office space now sits on a desktop and displays images that are helpful and amusing. Computers have become an integral part of business, communication, entertainment, and scientific research. Easy to use and increasingly more powerful, today's personal computers are the useful tools an infant industry dreamed they'd be.

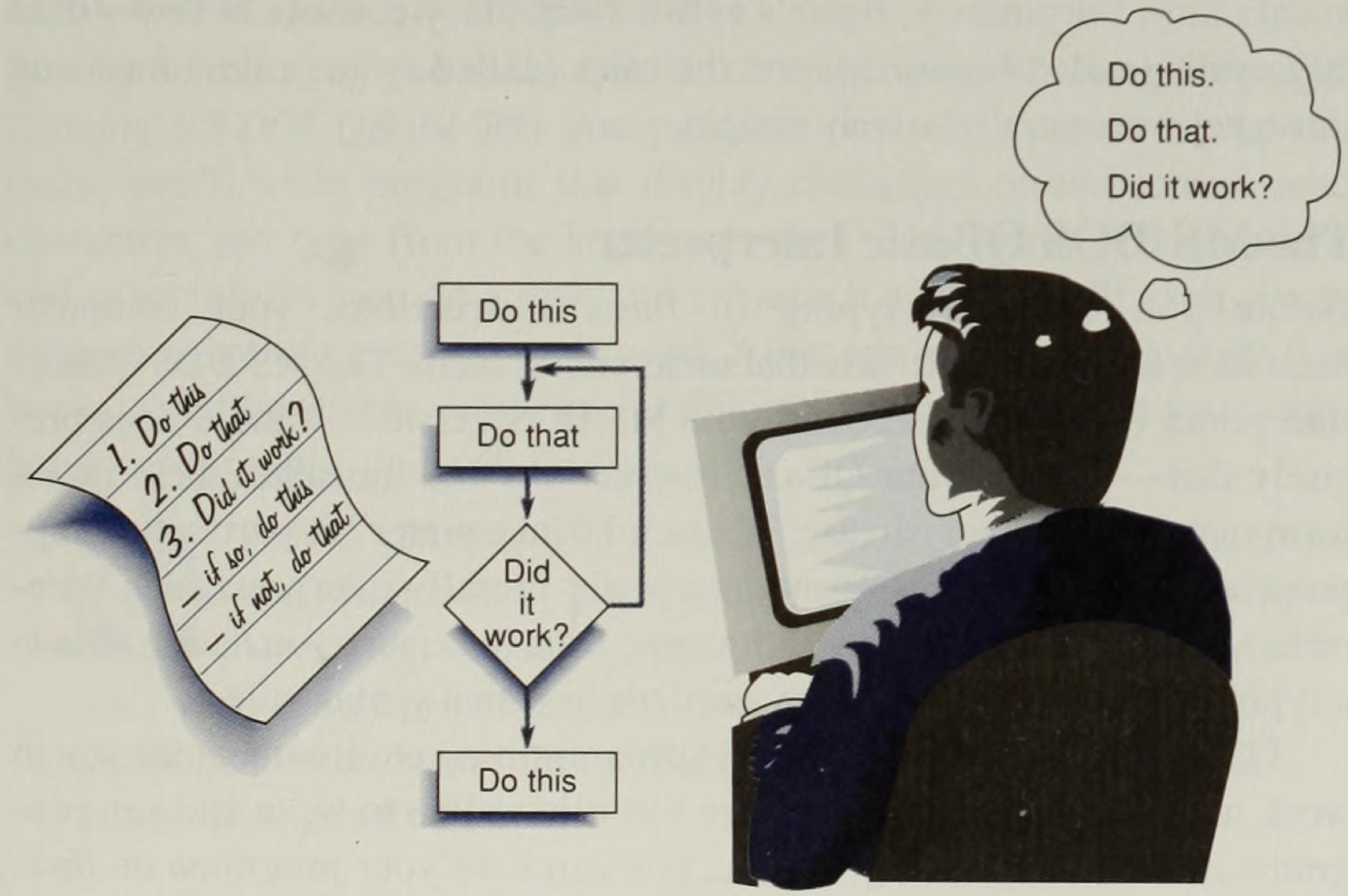
HOW DO I WRITE A COMPUTER PROGRAM?

Preparation and organization are the important first steps in the programming process. Two questions help you get organized:

1. What do I want this program to do?
2. What steps must the program take to do this?

The set of general steps you come up with is called an *algorithm.* An algorithm isn't a computer program itself; it's just a collection of ordered notes describing what your program will do each step of the way. In fact, if you've ever used a recipe you've used an algorithm. Baking a cake, for example, involves following a step-by-step process that produces a useful (and tasty!) result.

Because an algorithm is a problem-solving tool that you design to help you get the program written, there's no one right way of putting one together. Some programmers write an algorithm as a list of ordered steps on a piece of paper, some show the steps graphically in a diagram called a flowchart, and others (who have been working at it for some time) simply organize the programming steps in their heads.



A list A flowchart An experienced programmer

Algorithms: There's more than one way to create them.

Whichever method you choose, begin to think about your program in general terms well before you sit down in front of the computer.

The Basic Programming Language

Part of the reason for the emphasis on up-front problem solving and design is that the computer doesn't know very many words: You must spell things out for it in a language it understands.

Basic is one of several programming languages available for personal computers. As do spoken languages, each computer language has its own ''vocabulary'' and its own set of rules. In addition, each language has strong points and weak points. Some languages are more efficient at doing certain tasks than others; some are easier to work with than others. All things considered, Basic is an excellent language for computer programming because it's powerful, flexible, and—above all—English-like and easy to use.

Learning to ''speak'' Basic, or any other computer programming language, is very much like learning to speak a foreign language. Basic has its own vocabulary and a set of rules you must follow when using that vocabulary. Fortunately, Basic's entire vocabulary consists of fewer than 200 words (called *keywords*), and the rules (called *syntax* rules) for using those keywords are relatively simple.

The MS-DOS QBasic Interpreter

Before you can start typing in Basic instructions, your computer must have a piece of software that understands them. The MS-DOS QBasic interpreter (QBasic) that comes with MS-DOS versions 5 and 6 does precisely that—it reads your QBasic instructions and translates them into a form that the computer can use. The word ''interpreter'' is particularly appropriate: The MS-DOS QBasic interpreter literally interprets each statement as you type it, checking at the same time to be sure you haven't made a typing mistake, misused a keyword, or broken a syntax rule.

QBasic also provides a complete programming environment for you to work in. QBasic is a single package that allows you to write and run programs immediately. Using QBasic, you can save your programs on disk, print them on your printer, and get help when you need it. QBasic's environment even contains many word-processor-like features that make writing and changing your programs easy.

The Origins of QBasic

QBasic is a modern superset of the original BASIC language, created by John G. Kemeny and Thomas E. Kurtz in 1963 and 1964 at Dartmouth College, and of other versions of BASIC for the IBM Personal Computer and compatibles (such as BASICA and GW-BASIC). Not only is QBasic a superset of these older versions of BASIC—that is, it understands all of the older versions' keywords and syntax—but it also provides keywords and features from other popular computer languages, making it a complete system for writing your own programs.

A more advanced version of Basic, Microsoft Visual Basic for MS-DOS, is also available from Microsoft. We tell you a little about this language product in Chapter 13.

HOW WILL THIS BOOK HELP ME LEARN QBASIC?

Besides introducing you to the QBasic computer programming language, *Running MS-DOS QBasic* lets you put your new skills to work. As you learn, you'll write programs that display characters on the screen, read characters you type from the keyboard, work with words and numbers, and store information on a disk and retrieve it again. You'll even create programs that use graphics and sound. After you've worked through this book, you'll be able to create useful QBasic programs that put your computer—your tool—to work for you.

NOTE: Starting with Chapter 2, each chapter of this book ends with review questions and exercises. Take the time to work them through—they're a good opportunity to review what you've just learned and to be sure you understand it before you go on to the next chapter. You'll find answers to the questions and exercises in Appendix E.

QBASIC IS PART OF MS-DOS

QBasic comes free with MS-DOS versions 5 and 6. If you have either of these versions on your computer, you have QBasic. Before you get started, be sure that you have MS-DOS version 5 or 6 installed on your computer (if you have a hard disk) or that you have the MS-DOS working disks (if you have a two-floppy system with no hard disk). The next chapter gives you step-by-step instructions for starting QBasic and writing your first program!

NOTE: We assume that you have some elementary experience with the MS-DOS operating system (formatting disks, changing directories, setting paths, and so on). If you need to brush up on the basics, see the DOS documentation that came with your computer or consult Running MS-DOS *by Van Wolverton (Microsoft Press, 1993).*

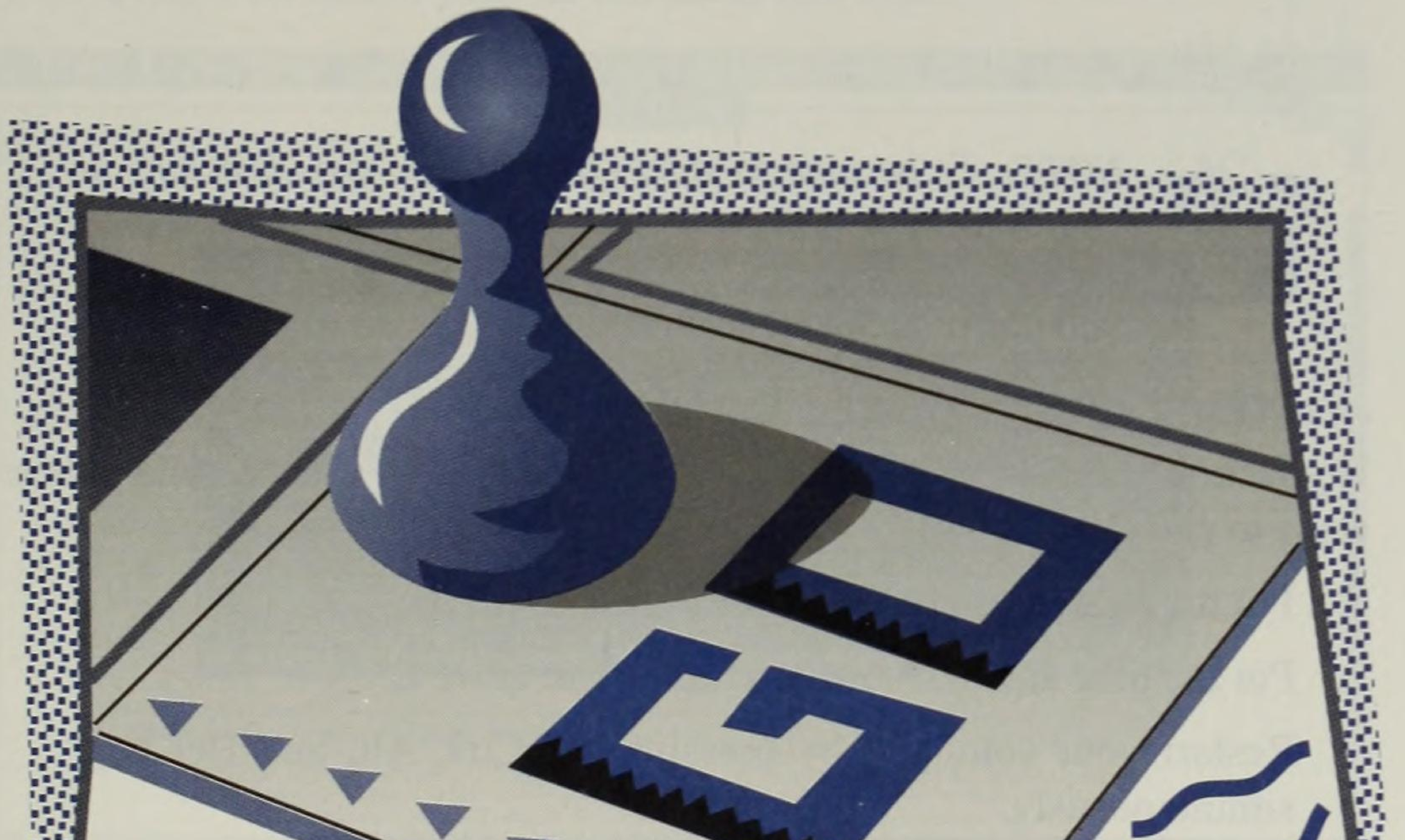
Appendix A: Your Reference to QBasic

In the following chapters, we assume that you're somewhat familiar with menu-based environments such as MS-DOS EDIT (the MS-DOS text editor) or with Microsoft Works (a multipurpose business application). If you need to review using the mouse or keyboard to navigate in a menu-driven environment, or if you need specific information about customizing the QBasic environment, see Appendix A, ''Using QBasic Menus and Options.'' Appendix A contains information on these topics:

- Selecting menus and commands
- Using dialog boxes
- Using QBasic keyboard commands
- Selecting text
- Getting online help
- Printing
- Changing display colors
- Using startup options

CHAPTER 2

Getting Started with QBasic



Now that you're prepared for the basics, let's get right down to business and start learning how to program.

STARTING QBASIC

To start to program, you must start QBasic. You do this in one of two ways, depending on whether you'll run QBasic from a hard disk or a floppy disk.

NOTE: Throughout this book, text that you (or your users) must enter from the keyboard appears in `color`.

Starting QBasic from a Hard Disk

If you have MS-DOS version 5 or 6 installed on a hard disk, the QBasic program and support files should be located in the DOS directory in drive C. To start QBasic, type the following commands, pressing the Enter key after each one:

```
c:
cd \dos
qbasic
```

NOTE: To be able to start QBasic from any directory on your hard disk, add the C:\DOS directory to the PATH environment variable stored in your AUTOEXEC.BAT file.

Starting QBasic from Floppy Disks

If your computer has two floppy-disk drives and no hard disk, follow these steps to run QBasic:

1. Put the disk labeled *MS-DOS Startup* in drive A.
2. Put the disk labeled *MS-DOS QBasic 1* in drive B.
3. Restart your computer by pressing the Ctrl, Alt, and Del keys simultaneously.
4. When the system prompt appears, type the following PATH command and then press the Enter key:

   ```
   path a:;b:
   ```

5. Start QBasic by typing the following command and then pressing the Enter key:

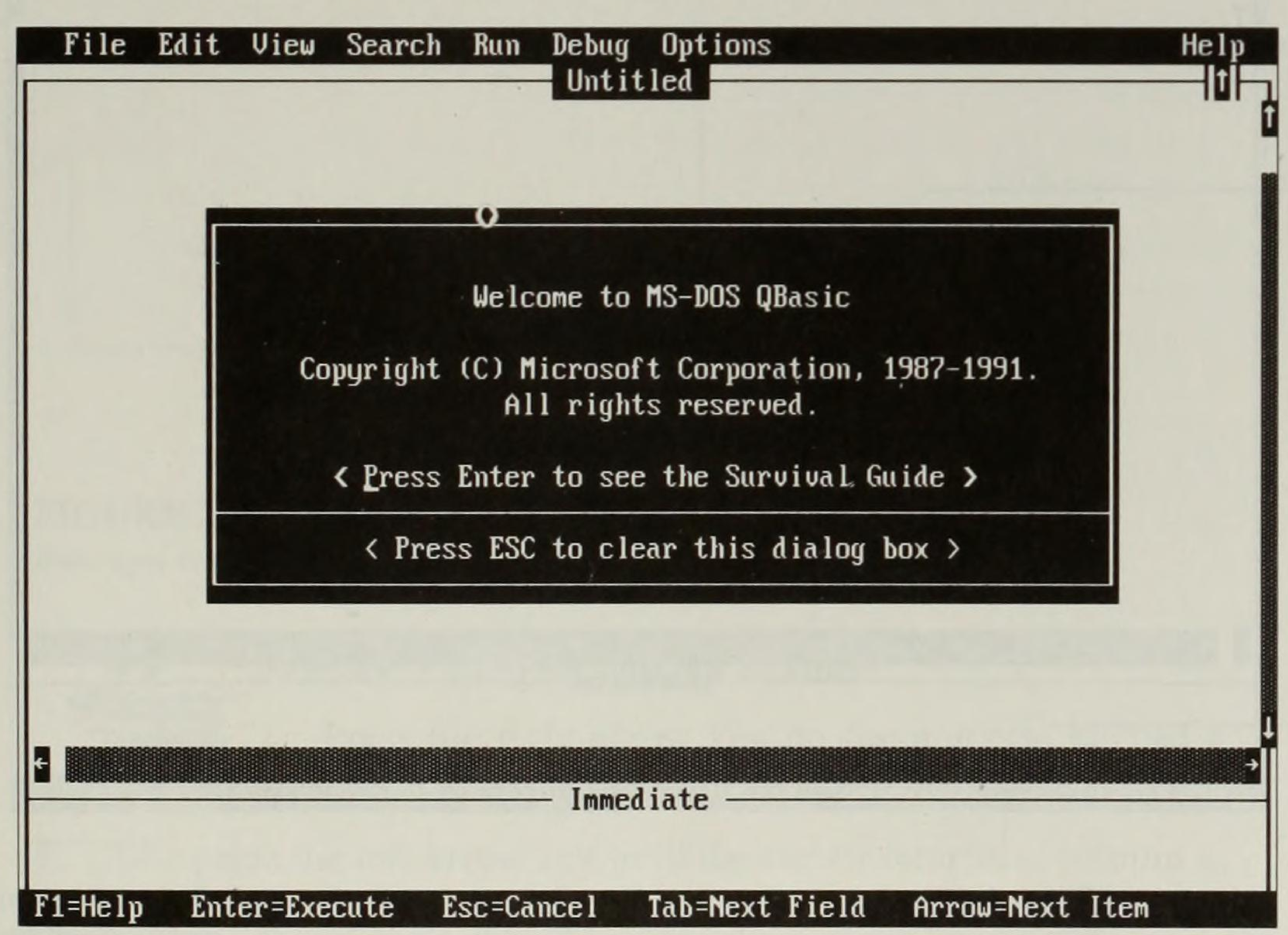
   ```
   qbasic
   ```

You will see the drive B light go on, and QBasic will appear on the screen. When the drive lights go out, remove the *MS-DOS Startup* disk from drive A and insert the *MS-DOS QBasic 2* disk in drive A. Run QBasic from both QBasic disks until you quit the program. Then remove the *MS-DOS QBasic 2* disk from drive A and insert the *MS-DOS Startup* disk in drive A.

> *NOTE: As your software programs get bigger, you'll find it increasingly difficult to run programs from a system with no hard disk. Your biggest problem with running QBasic will be running out of disk space and having to swap disks often. If you spend lots of time working with your computer, consider buying a hard disk or upgrading to a computer that has one.*

THE QBASIC SCREEN

After QBasic is loaded, your screen looks like this:



File
Edit
View
Search
Run
Debug
Options
Help
Untitled
Immediate
F1=Help
Enter=Execute
Esc=Cancel
Tab=Next Field
Arrow=Next Item

Welcome to MS-DOS QBasic

Copyright (C) Microsoft Corporation, 1987-1991.
All rights reserved.

< Press Enter to see the Survival Guide >

< Press ESC to clear this dialog box >

This "Welcome" message appears each time you start QBasic. The screen offers you two options:

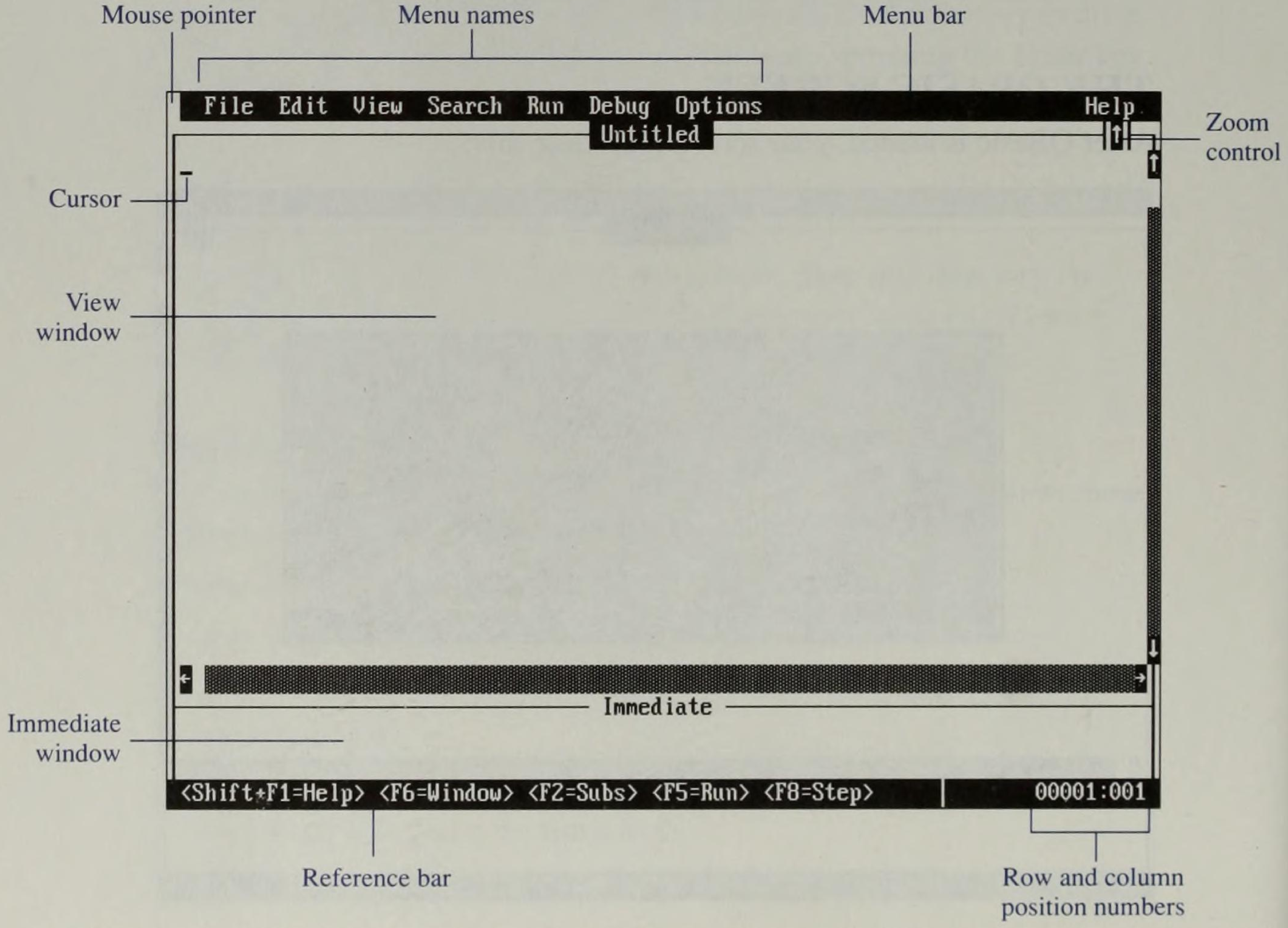
- You can press Enter to see QBasic's built-in help system.
- You can press the Esc key to clear the message from the screen and start to program.

Practice: Clearing the screen to begin programming

Because you will be programming in this chapter, press Esc to clear the message from the screen.

The QBasic Environment

Let's begin with a look at the various elements of the QBasic screen that will help you create programs and process text.


Mouse pointer
Menu names
Menu bar
File Edit View Search Run Debug Options
Help
Untitled
Zoom control
Cursor
View window
Immediate
Immediate window
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00001:001
Reference bar
Row and column position numbers

The cursor

The blinking underscore in the upper-left corner of the screen will probably catch your eye first. This is the *cursor*. The cursor indicates where the next character you type will appear on the screen.

The mouse pointer

If you have a mouse installed on your computer, a rectangular *mouse pointer* appears in the upper-left corner of the screen. The mouse pointer lets you move around your program quickly and choose menu items easily.

The row and column position numbers

In the lower-right corner of the screen are two numbers separated by a colon. These numbers indicate the current location of the cursor. The first number tells you which row the cursor is in; the second number (following the colon) tells you which column the cursor is in. As shown in Figure 2-1, the numbers indicate that the cursor is in row 1, column 1.

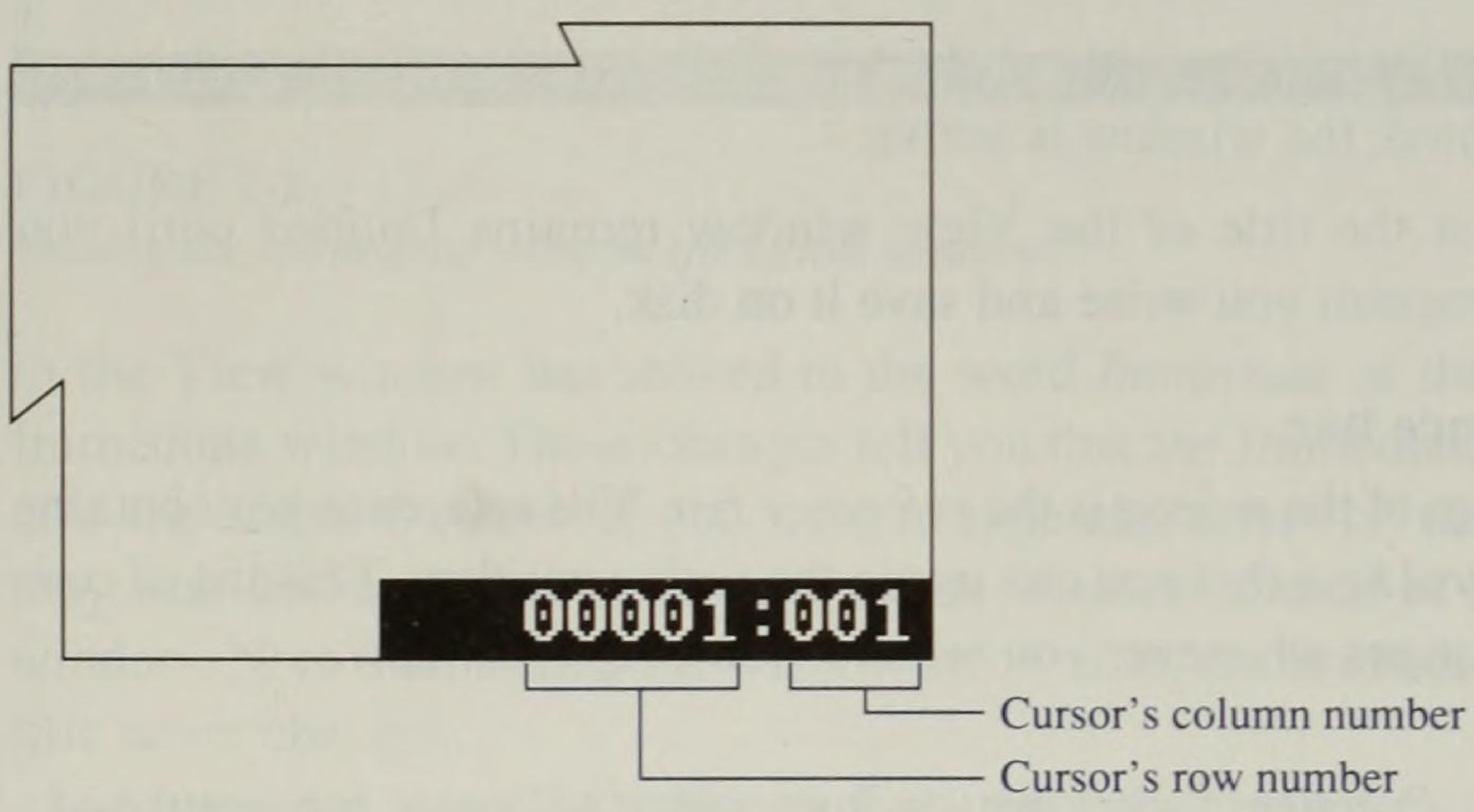


FIGURE 2-1.
Row and column position numbers.

Practice: Changing column numbers

1. Press the right-arrow key on the numeric keypad a few times, and watch what happens to the column number.
2. Now press the left-arrow key until the cursor returns to column 1. (Later, when you begin typing in programs, you can practice changing row numbers with the up-arrow and down-arrow keys.)

The menu bar and the menu names

Across the top of the screen is the *menu bar*. The menu bar contains the *menu names* for all QBasic drop-down menus.

The View window and the Immediate window

QBasic initially gives you two windows to work in:

- The upper window—where you type in and work with your own programs—is called the *View window*.
- The lower window—where you can test out a programming instruction before you actually type it into your program—is called the *Immediate window*.

You can work in only one window at a time, although it's easy to switch between the two. The window you're currently working in is called the *active window*. Identifying the active window is easy:

- Look for the blinking cursor; it's always in the active window.
- Look at the window title at the top of the window; if the title is highlighted, the window is active.

Note that the title of the View window remains *Untitled* until you name the program you write and save it on disk.

The reference bar

At the bottom of the screen is the *reference bar*. The reference bar contains a list of *control keys* that you can use in the active window. This list of control keys changes whenever you select a new active window.

Practice: Switching windows with a reference bar command

Notice that the F6 function key is the Window key. You use F6 to switch between windows. (To switch between windows with the mouse, simply move the mouse pointer anywhere within the window you want to be active and click.) Press F6 now and see what happens.

As shown in Figure 2-2, the blinking cursor is now in the Immediate window. The shaded horizontal and vertical scroll bars have disappeared from the View window, and the highlight that was on the word *Untitled*

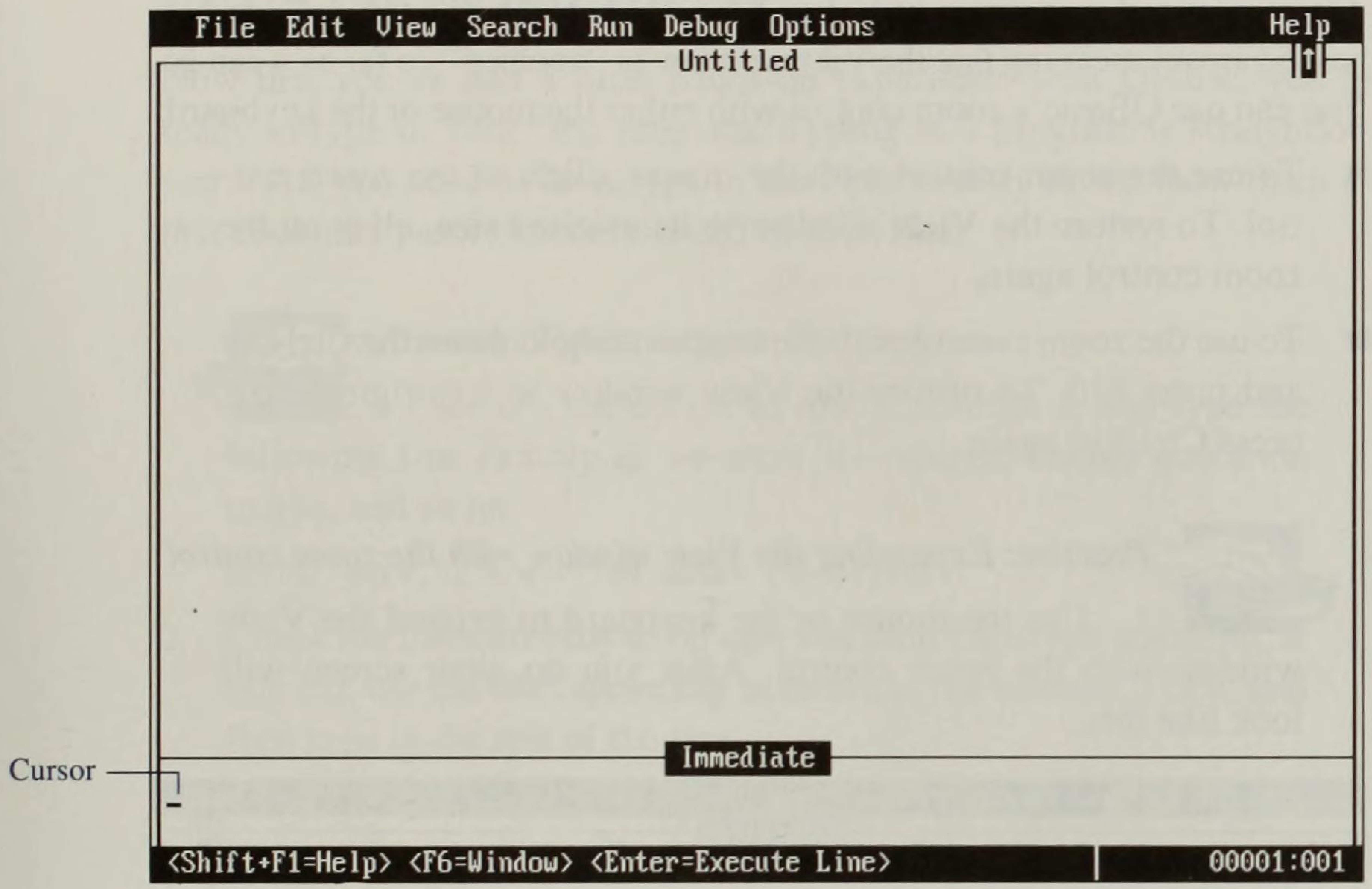


FIGURE 2-2.
Making the Immediate window the active window.

in the View window has moved to the word *Immediate* at the top of the Immediate window. These changes tell you that the Immediate window is now the active window. If you were to type characters on the keyboard, they would now appear in the Immediate window instead of in the View window. Note that, unlike the View window title, the Immediate window title never changes.

Also notice that a different set of control keys now appears in the reference bar at the bottom of the screen. These are the Immediate window's control keys. For now, you won't be using the Immediate window, so press F6 to make the View window active again.

The zoom control

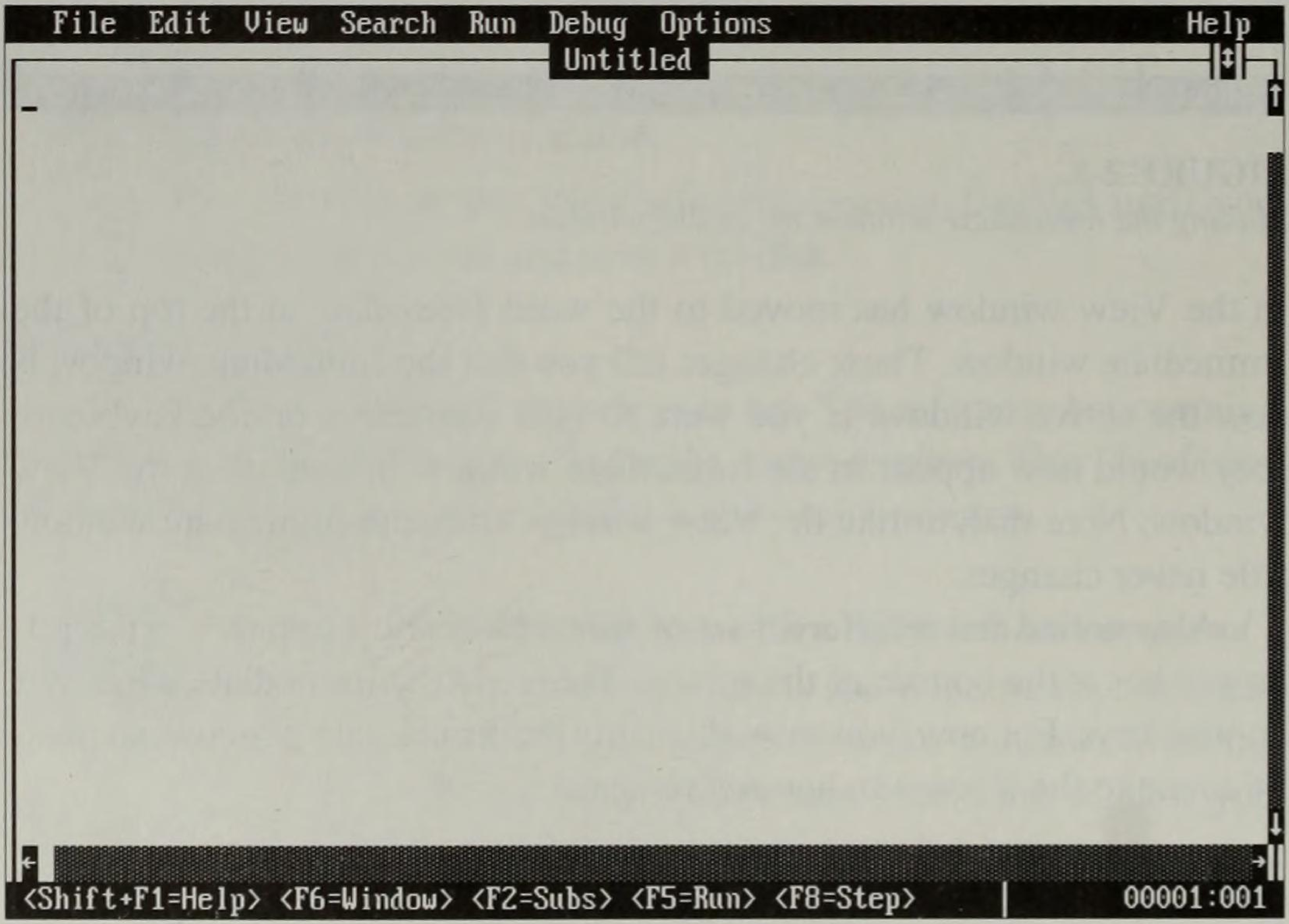
You can make the Immediate window disappear from view. In the upper-right corner of the screen, just under the Help menu name, is a vertical rectangle with an up-arrow character. This is the *zoom control,* which causes the View window to expand and cover the Immediate window.

After you use the zoom control, the up-arrow character changes to a double-headed arrow, meaning that the View window is "zoomed" as far as it can go. You can use QBasic's zoom control with either the mouse or the keyboard:

- To use the zoom control with the mouse, click on the zoom control. To restore the View window to its original size, click on the zoom control again.
- To use the zoom control with the keyboard, hold down the Ctrl key and press F10. To restore the View window to its original size, press Ctrl-F10 again.

Practice: Expanding the View window with the zoom control

1. Use the mouse or the keyboard to expand the View window with the zoom control. After you do, your screen will look like this:



2. Now restore the View window to its regular size to make the Immediate window visible.

YOUR FIRST PROGRAM

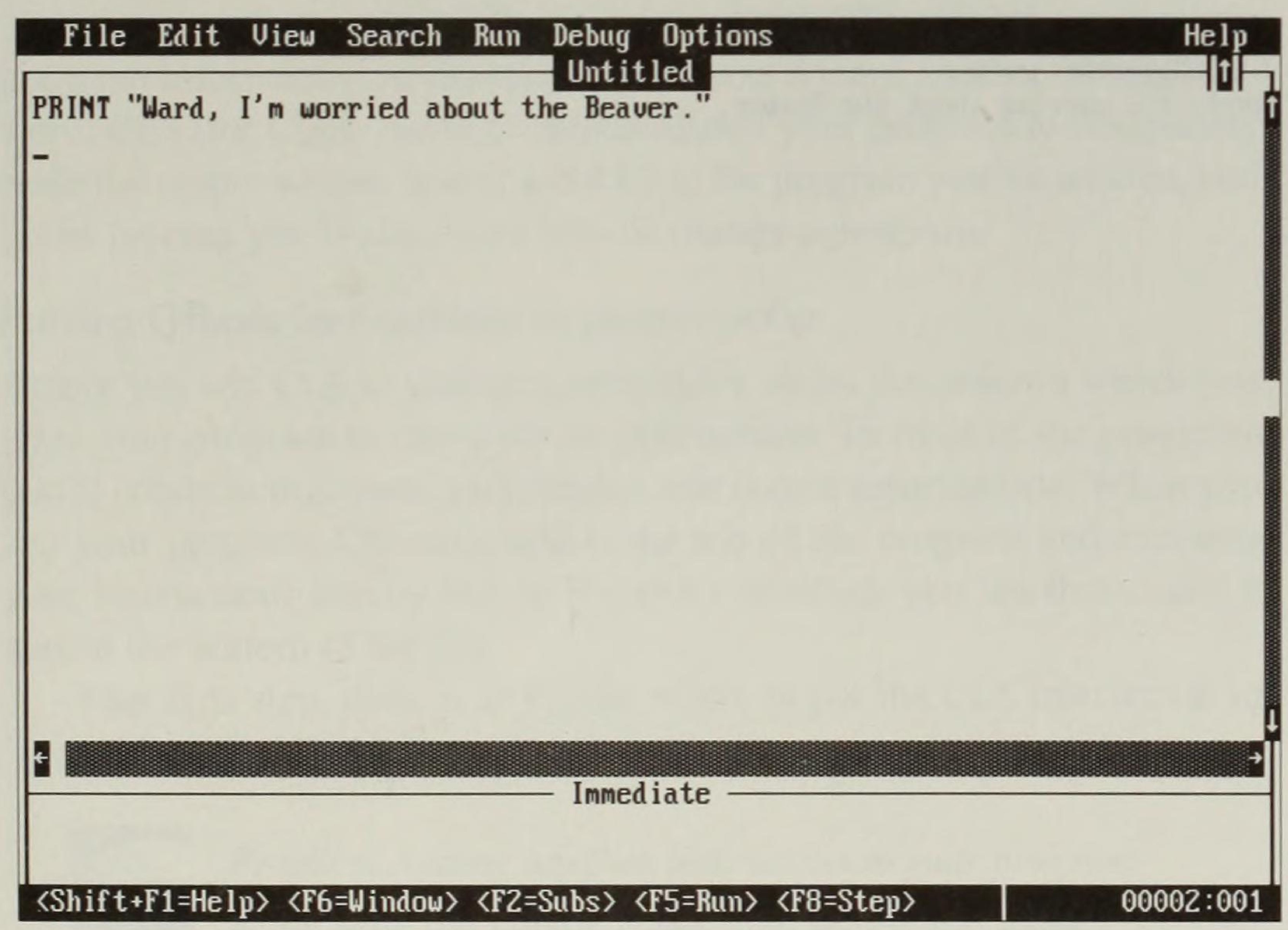
Now that you've had a little hands-on experience with QBasic, you're ready to type in your first program. Typing in a program is straightforward. All you need to do is type in the lines exactly as we show them in this book and press Enter at the end of each line.

Practice: Typing in your first program

1. Return the cursor to row 1, column 1, and type the following line exactly as we show it—spaces, double quotation marks, and so on:

   ```
   print "Ward, I'm worried about the Beaver."
   ```

2. Check the line carefully to be sure you didn't mistype anything. If you did, use the Backspace key to return to the mistake. Fix it, and then type in the rest of the line.
3. When you're satisfied that you haven't made any mistakes, press Enter. After you press Enter, your screen looks like this:



File Edit View Search Run Debug Options Help
Untitled
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00002:001

```
PRINT "Ward, I'm worried about the Beaver."
```

Notice that the word *print* is in capital letters now because the word *print* is a QBasic *keyword*—that is, it's a part of QBasic's "vocabulary." QBasic always converts keywords to all uppercase so that you can easily distinguish them from everything else on a line.

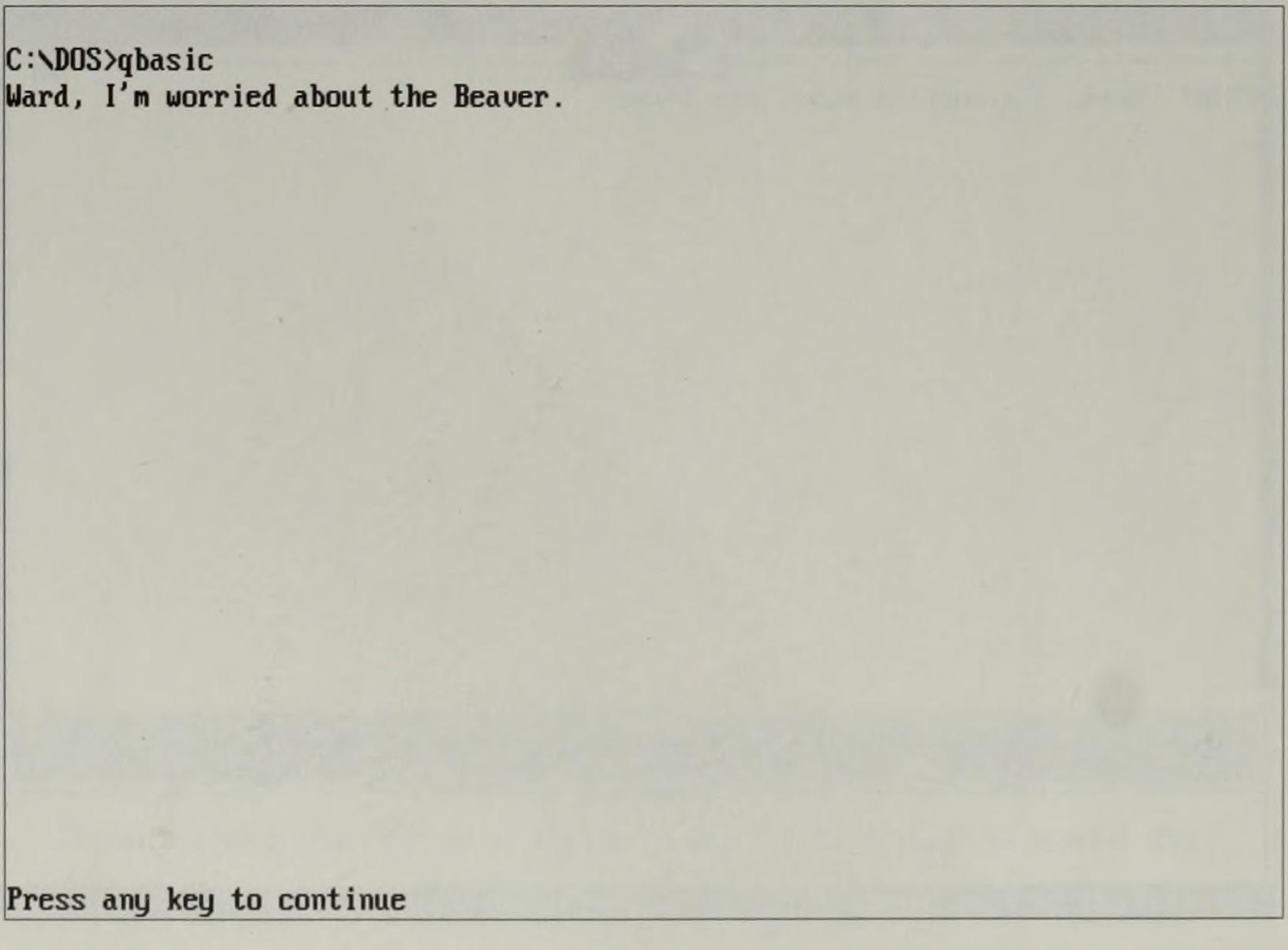
What you just typed in, although it's only one line long, is a complete QBasic program.

Running a Program

To run a program, you choose the Start command from the Run menu. Doing this tells QBasic to execute the instructions currently in the View window.

Practice: Running your first program

Run your program now: Press Alt to activate the menu bar, choose Run, and then choose Start. (To review using the keyboard or the mouse to select menu items, see Appendix A, "Using QBasic Menus and Options.") Your screen looks something like this:



```
C:\DOS>qbasic
Ward, I'm worried about the Beaver.
















Press any key to continue
```

The Output Screen

QBasic displays the output of the program, the end result of the instructions in your program, on the *output screen.* When you run the program, QBasic switches immediately to the output screen to prevent your program's output from writing over the program instructions (or *code*) in the View window.

The output screen displays the output from any program that you have run since the last time you cleared the screen. That's why you see the DOS command line you used to start QBasic. The message at the bottom of the output screen, *Press any key to continue*, appears every time a program finishes running.

Practice: Returning to the View window

Press any key now. QBasic returns you to the View window. Congratulations! You've written and run your first BASIC program!

Changing Your Program

You've learned about one QBasic keyword, PRINT, which lets your program put information on the screen. Now you'll learn another QBasic keyword, CLS (for CLear Screen), which allows your program to completely erase the output screen. You'll add CLS to the program you've written, and in the process you'll also learn how to change a program.

Putting QBasic instructions in proper order

Before you add CLS to your program, think about the order in which you want your program to carry out its instructions. In most of the programs you'll create in this book, each instruction is on a separate line. When you run your program, QBasic starts at the top of the program and executes your instructions one by one in the order in which you list them until it gets to the bottom of the list.

Your first step, then, is to decide where to put the CLS instruction in your program.

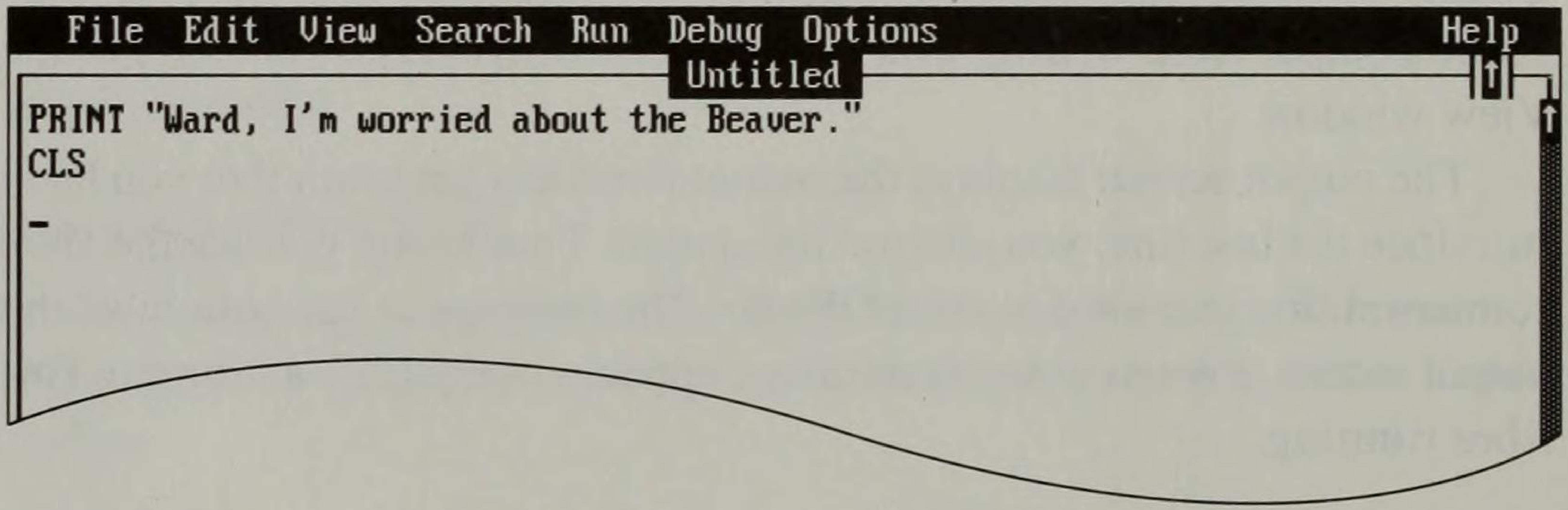
Practice: Adding another instruction to your program

Right now the cursor is one line below the letter P of your PRINT instruction.

1. Type the following and press Enter:

   ```
   CLS
   ```

   Your screen should now look like this:



```
PRINT "Ward, I'm worried about the Beaver."
CLS
```

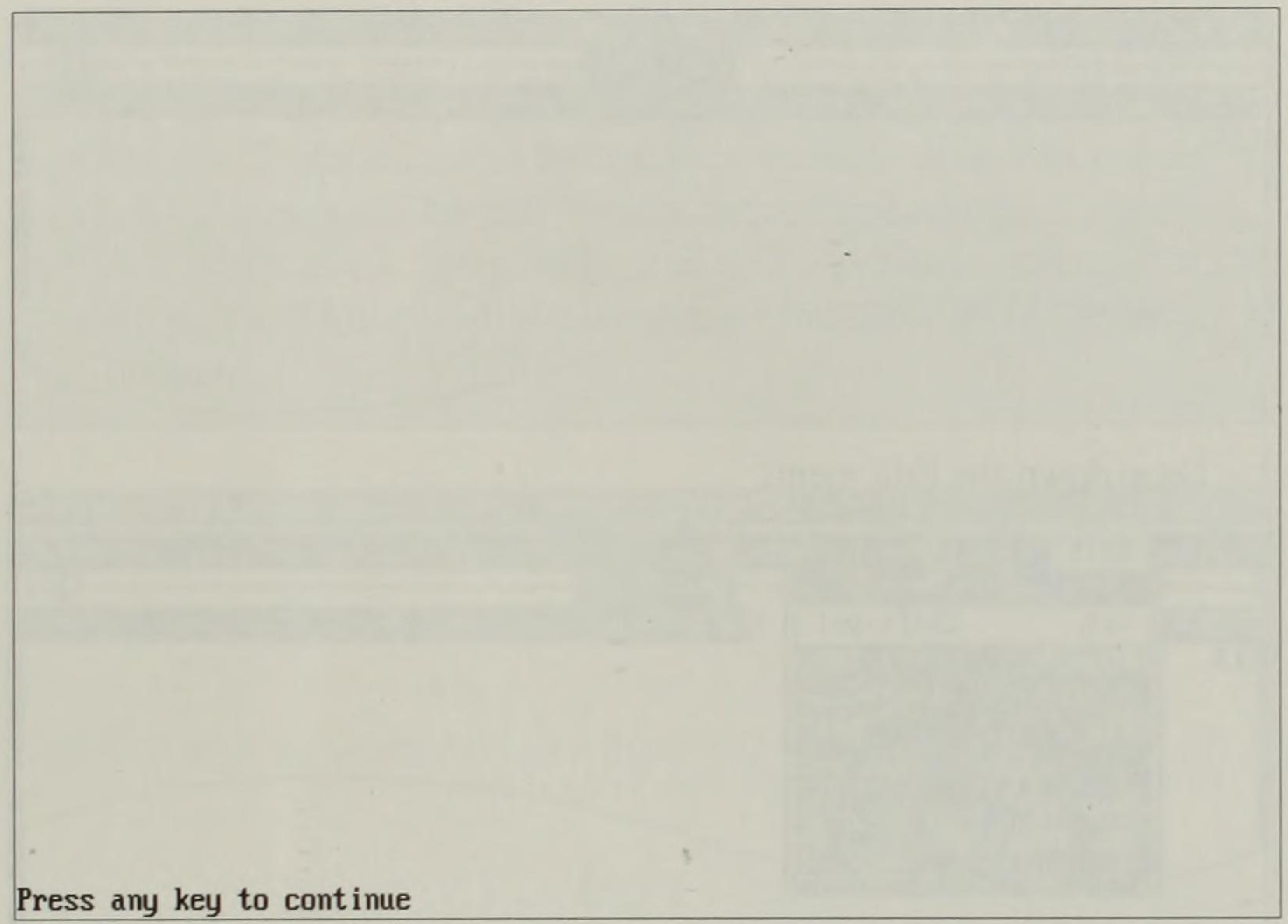
   (As before, if you typed in the CLS instruction in lowercase letters, QBasic converts them to uppercase letters after you press Enter because CLS is a QBasic keyword.)

2. Run the program again: Choose Start from the Run menu. If you're watching carefully, you might see the message *Ward, I'm worried about the Beaver* flash briefly on the screen, but a quick moment later, the output screen looks like the screen on the opposite page. There's little point in using a PRINT instruction if you're going to follow it immediately with a CLS instruction. Now you see why the order of the instructions is so important. You need to put the CLS instruction *before* the PRINT instruction to make the program behave the way you expect it to.
3. Press any key to return to the View window.

Editing a Program

When you change a program, you perform a process known as *editing*. Editing includes adding characters, deleting characters, and moving lines of instructions to other parts of your program.

QBasic gives you several editing tools. Some of these tools are on the aptly named Edit menu; others are keys on your keyboard. Use the following practices as an overview. If you find that you need more information, turn to Appendix A, "Using QBasic Menus and Options," which contains detailed instructions on how to use these tools.



```
Press any key to continue
```

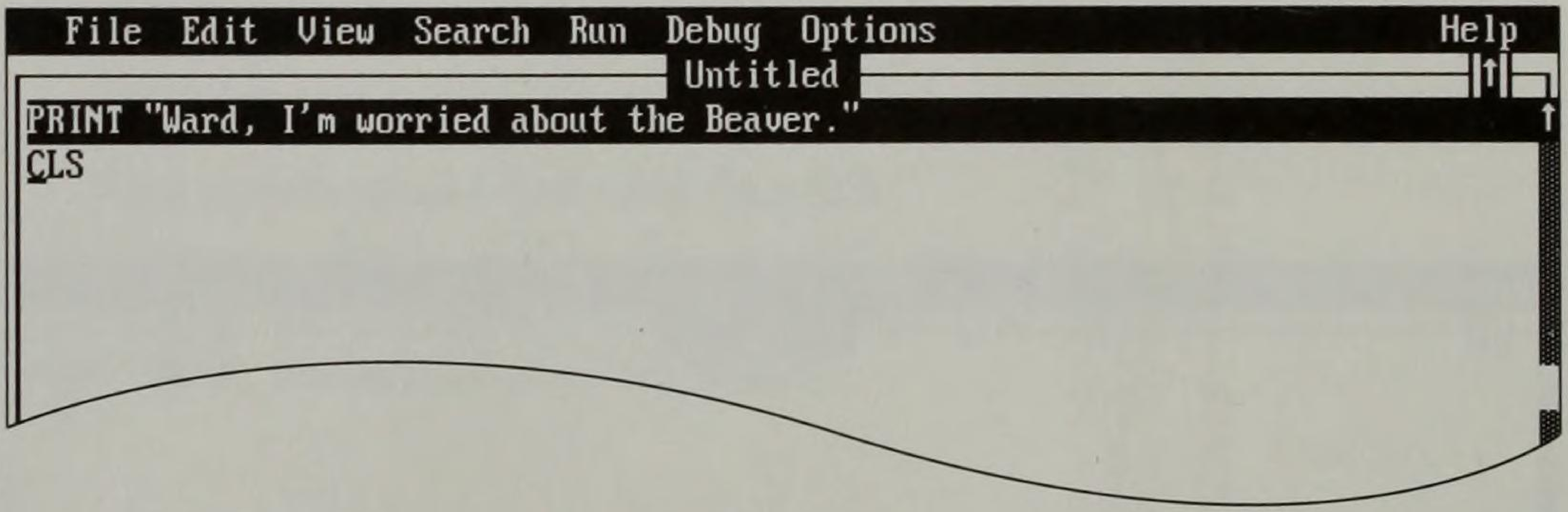
The Cut, Copy, and Paste commands

The Cut, Copy, and Paste commands let you edit your programs efficiently. Not only do they save you time and lots of typing, but they also reduce the chances of your making typing errors.

Practice: Editing a program with the Cut and Paste commands

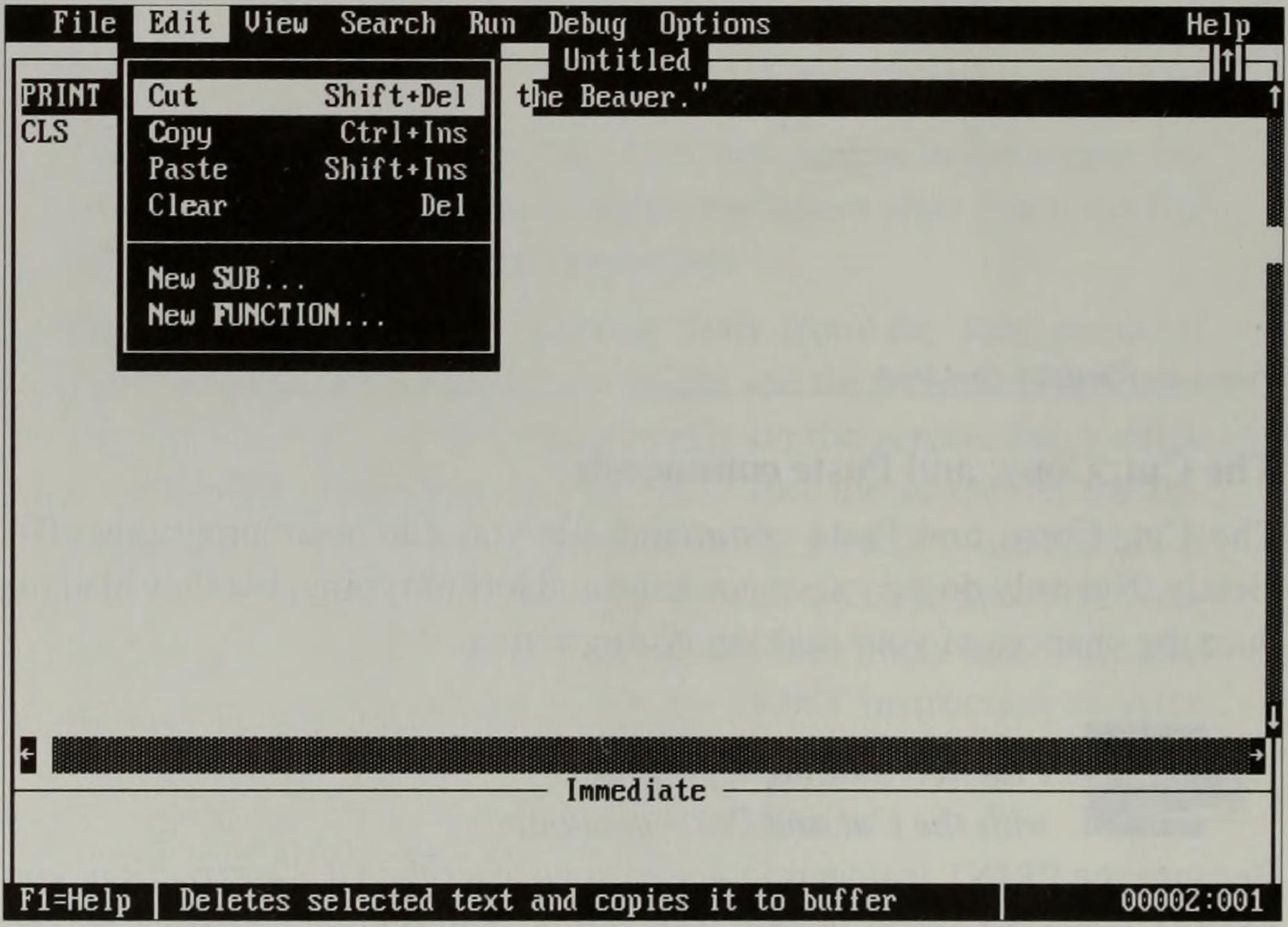
Because the PRINT instruction needs to go after the CLS instruction, you need to move the entire PRINT instruction. You do this by using the Cut and Paste commands on the Edit menu.

1. Move the cursor under the P in PRINT. Next, hold down the Shift key and press the ~~down~~ up-arrow key to select the PRINT instruction line. Your screen should look like the one on the next page.


File Edit View Search Run Debug Options Help
Untitled

```
PRINT "Ward, I'm worried about the Beaver."
CLS
```

2. Drop down the Edit menu:


File Edit View Search Run Debug Options Help
Untitled
Cut Shift+Del
Copy Ctrl+Ins
Paste Shift+Ins
Clear Del
New SUB...
New FUNCTION...
Immediate
F1=Help

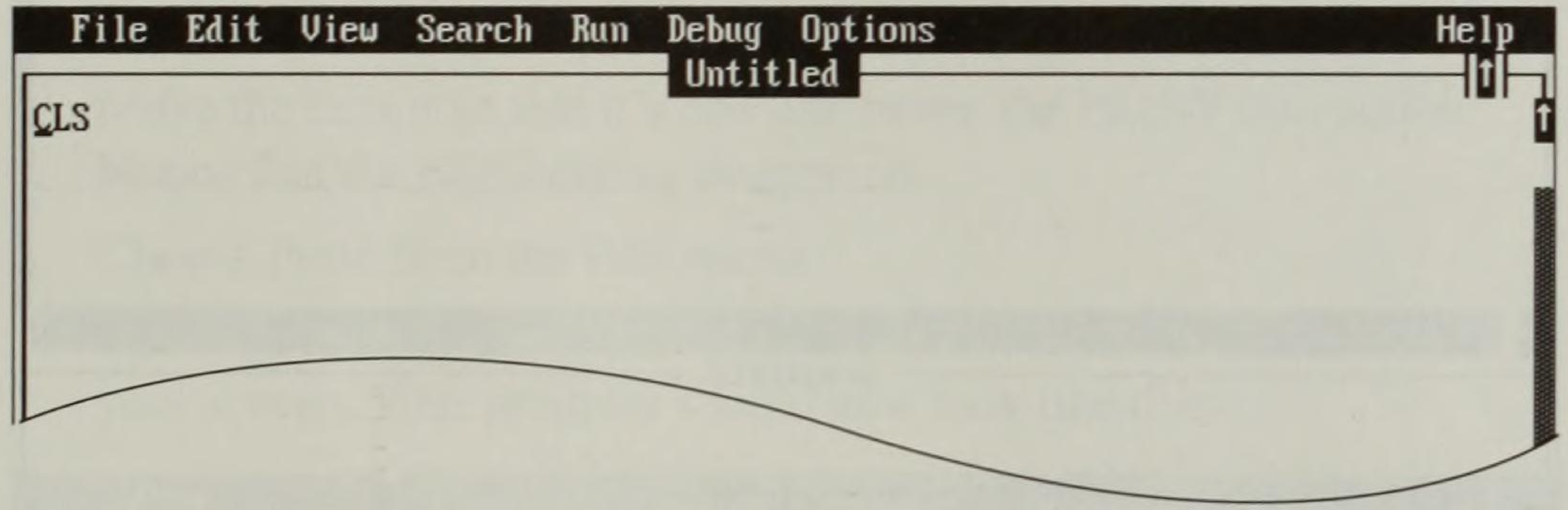
```
PRINT                   the Beaver."
CLS
```

```
F1=Help | Deletes selected text and copies it to buffer        00002:001
```

3. Choose the Cut command from the Edit menu. The PRINT instruction should disappear from the View window, leaving you with only the CLS instruction, as shown on the opposite page. Although it's no longer on the screen, the PRINT instruction isn't really gone. It's now in QBasic's *Clipboard.* Your next step is to paste the contents of the Clipboard back into your program.

The Clipboard

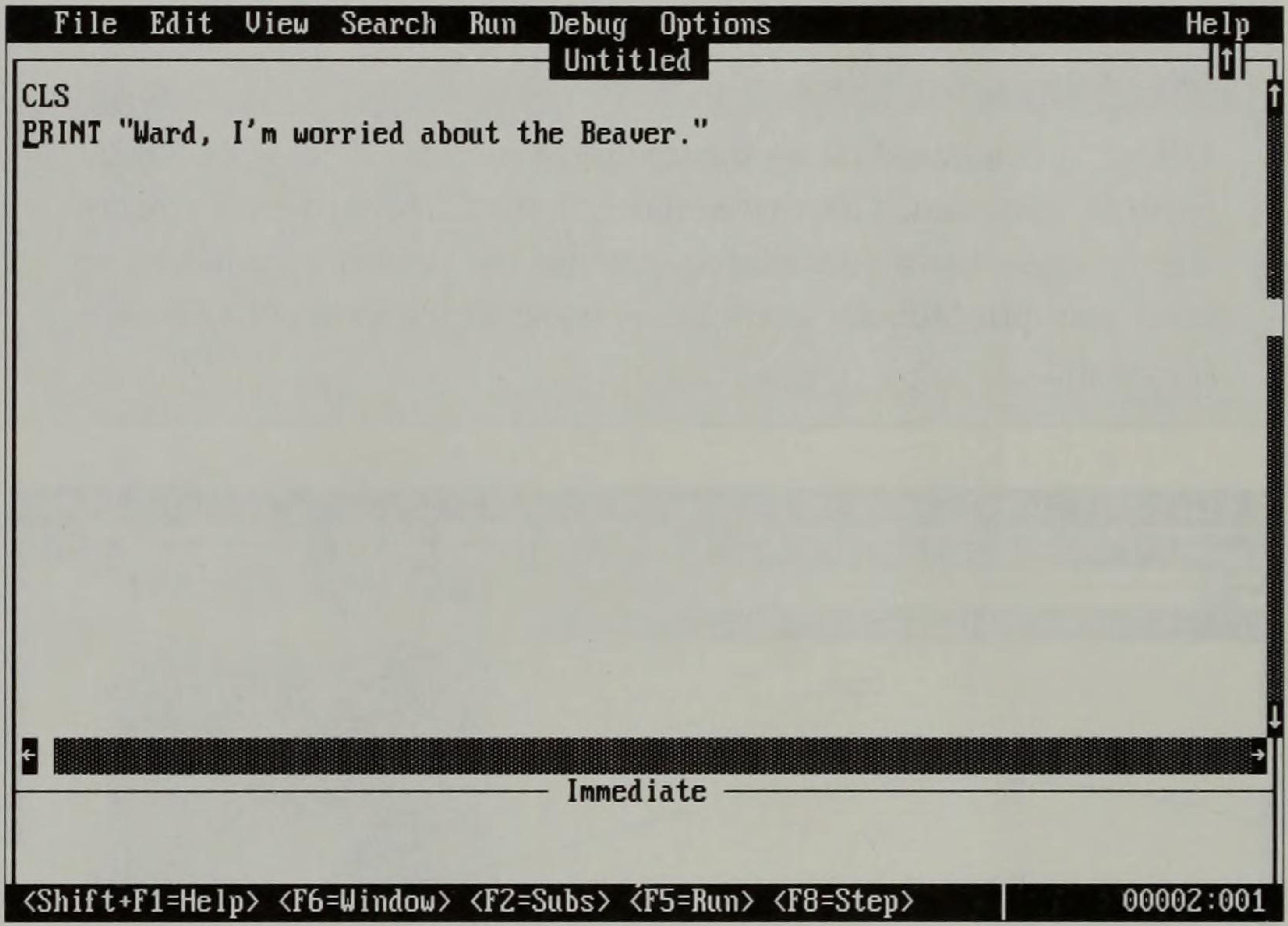
QBasic's Clipboard stores the text most recently deleted or copied from the program. This text remains in the Clipboard until you delete or copy again (completely erasing the previous contents) or until you quit QBasic. You cannot erase the contents of QBasic's Clipboard.



```
CLS
```

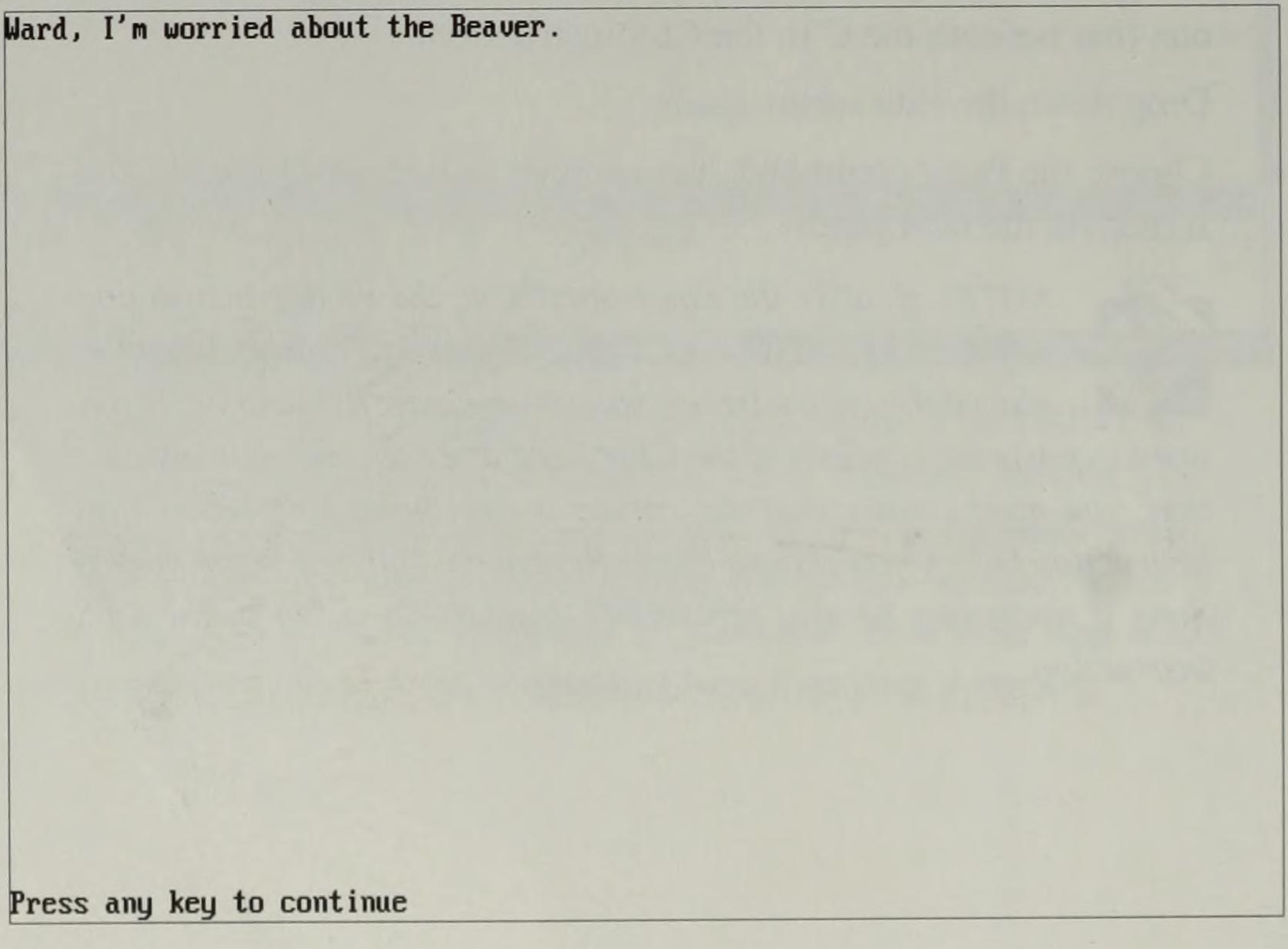
4. Move the cursor to the location at which you want to paste the text. Because you want to put the PRINT instruction line after the CLS instruction line, press the down-arrow key once to put the cursor one line beneath the C in the CLS instruction.
5. Drop down the Edit menu again.
6. Choose the Paste command. Your screen should now look like the screen on the next page.

NOTE: If, after the Paste operation, the PRINT instruction appears before the CLS instruction, you probably misplaced the cursor before you chose Paste. Remember: If you want to paste the contents of the Clipboard after a particular instruction, you must ensure that the cursor is one entire line below that instruction before you choose Paste. Repeat the Cut and Paste operations if necessary so that the PRINT instruction is below the CLS instruction.


File
Edit
View
Search
Run
Debug
Options
Help
Untitled
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00002:001

```
CLS
PRINT "Ward, I'm worried about the Beaver."
```

7. Run the program now to see what it does with the CLS instruction before the PRINT instruction:



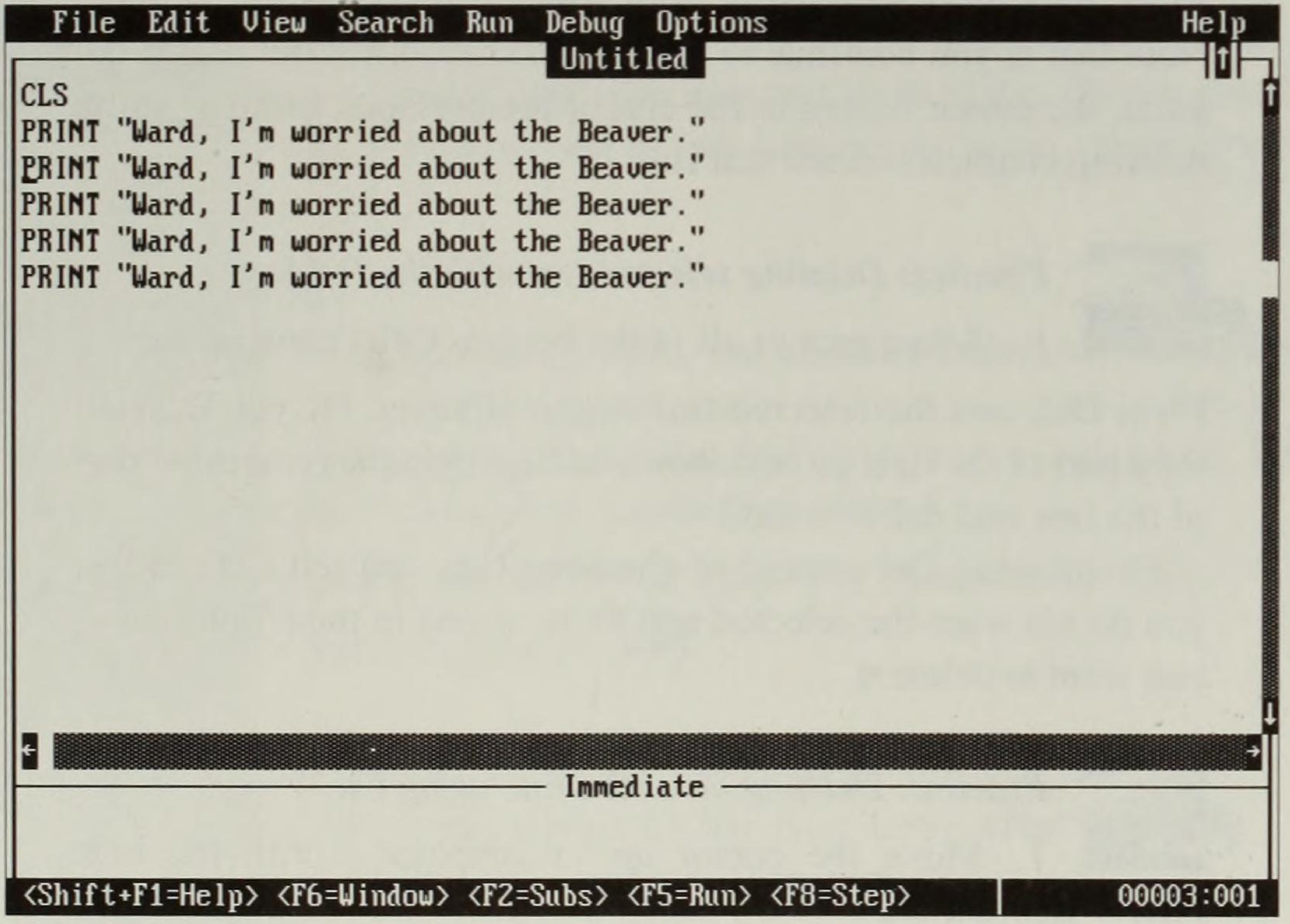
```
Ward, I'm worried about the Beaver.

Press any key to continue
```

8. Press any key to return to the View window.

Practice: Editing a program with the Copy and Paste commands

Now that you have the CLS instruction properly positioned, it's time to lengthen your program by using the Copy and Paste commands.

1. Select the entire PRINT instruction again.
2. Choose the Copy command from the Edit menu. Notice that the line you've selected remains in your program.
3. Move the cursor so that it's one line below the PRINT instruction. Notice that the highlighting disappears.
4. Choose Paste from the Edit menu.
5. Choose Paste again—until you have five Print instructions on your screen. Your program should now look like this:



```
CLS
PRINT "Ward, I'm worried about the Beaver."
PRINT "Ward, I'm worried about the Beaver."
PRINT "Ward, I'm worried about the Beaver."
PRINT "Ward, I'm worried about the Beaver."
PRINT "Ward, I'm worried about the Beaver."
```

Run the program again if you want to, but by now you're probably getting a little tired of seeing that same message over and over again. In fact, now would be a good time to practice deleting some of these instructions.

Practice: Deleting text with the Del key

1. Move the cursor to the P in the last PRINT instruction, and press the Del key several times. Notice how Del erases the character the cursor is under.
2. Keep pressing Del until the entire line has been deleted.
3. Press Del a few more times; eventually, you'll hear a beep each time you press Del. This beep is QBasic's way of telling you that there are no more characters to delete on that line. (Anytime you hear that sound, you're probably trying to do something QBasic won't allow.)

Practice: Deleting text with the Backspace key

1. Move the cursor up one line.
2. Press the End key to move the cursor to the end of the line.
3. Press Backspace repeatedly until the entire instruction is gone. Note that if you continue to press Backspace after the line is deleted, the cursor moves to the end of the previous line and starts deleting characters from that line.

Practice: Deleting selected text with the Del key

1. Select part or all of the bottom PRINT instruction.
2. Press Del, and the selected text is gone forever. (If you selected only part of the line, go back now and highlight the remaining part of the line and delete it too.)

   By pressing Del instead of choosing Cut, you tell QBasic that you do *not* want the selected text to be stored in the Clipboard—you want to delete it.

Practice: Deleting an entire line using Ctrl-Y

1. Move the cursor up to anyplace within the next PRINT instruction.

2. Hold down Ctrl and press Y. This has exactly the same effect as highlighting the entire line and choosing Cut from the Edit menu.

So much for the extra PRINT instructions! If you followed the instructions exactly, you're down to the CLS instruction and one PRINT instruction. If you have more instructions than that left, delete them now.

Changing Text

At times you might want to change a part of an instruction. You could simply delete the entire instruction and then retype it, but you'll find that changing only part of an instruction will save you time and reduce the chance of making a typing error.

Let's consider a PRINT instruction. A PRINT instruction prints whatever is between the two double quotation marks. If you've used a PRINT instruction in your program but decide to change the message, all you have to do is change the message between the double quotation marks. You can do this in several ways.

You could delete the existing text (using one of the methods of deletion that you just practiced) and type new text in its place. Or you could simply type in new text using one of two cursors: the *insert cursor* or the *overstrike cursor.*

The insert cursor

The cursor you've been working with, the blinking underscore, is called the *insert cursor.* It's called the insert cursor because you can use it to insert text in the middle of a line without destroying any existing text. To insert text with the insert cursor, follow these steps:

1. Move the cursor to the location at which you want to insert text.
2. Type in the text you want to insert.

Practice: Inserting text with the insert cursor

1. Move the cursor to the first letter after the first double quotation mark of your PRINT message.

2. Type a few letters and watch what happens. The letters currently in the message, including the letter the cursor is under, move to the right as you type, making room for the new letters to appear.
3. When you're done, use Backspace to delete the new characters you typed and restore your original message.

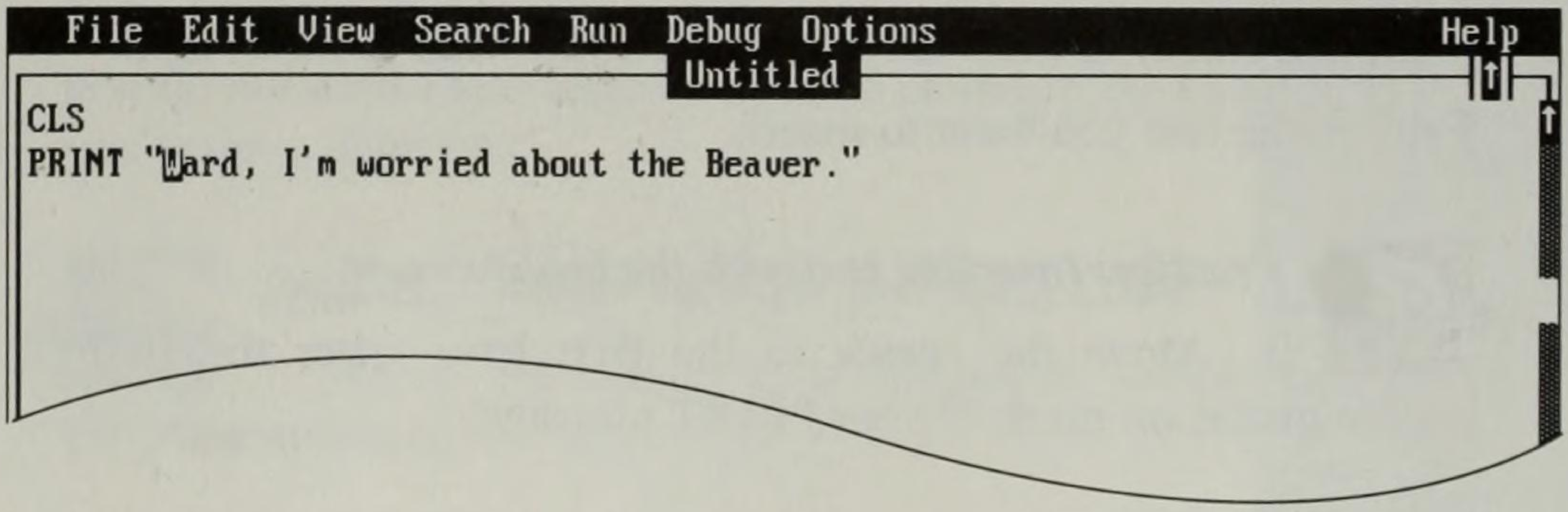
The overstrike cursor

The other cursor is called the *overstrike cursor.* It's called the overstrike cursor because it replaces existing characters with the new characters you type. Using the overstrike cursor can sometimes save you time because you don't have to delete characters as you would if you had only the insert cursor to work with. To use the overstrike cursor, follow these steps:

1. Move the cursor to the beginning of the characters you want to overwrite.
2. Press the Ins key. The cursor will change to a flashing rectangle.
3. Type the characters you want.

Practice: Using the overstrike cursor

1. Move the cursor to the first letter after the first double quotation mark of your PRINT message.
2. Press the Ins key. The cursor changes from a blinking underline to a blinking vertical rectangle:
3. Start typing a new message, and watch what happens. As you type, the new characters replace the old ones.



```
CLS
PRINT "Ward, I'm worried about the Beaver."
```

4. If your new message is shorter than the old message, use Del to delete the remaining characters (except for the closing double quotation mark). If your new message is longer than the old message, you will have overwritten the closing double quotation mark, so be sure to type a new one.
5. Run the program again to display your new message, and then press any key to return to the View window.
6. Press Ins to change the cursor from overstrike mode back to insert mode. You'll probably want to leave the cursor in insert mode as you work through this book.

Saving Your Program on Disk

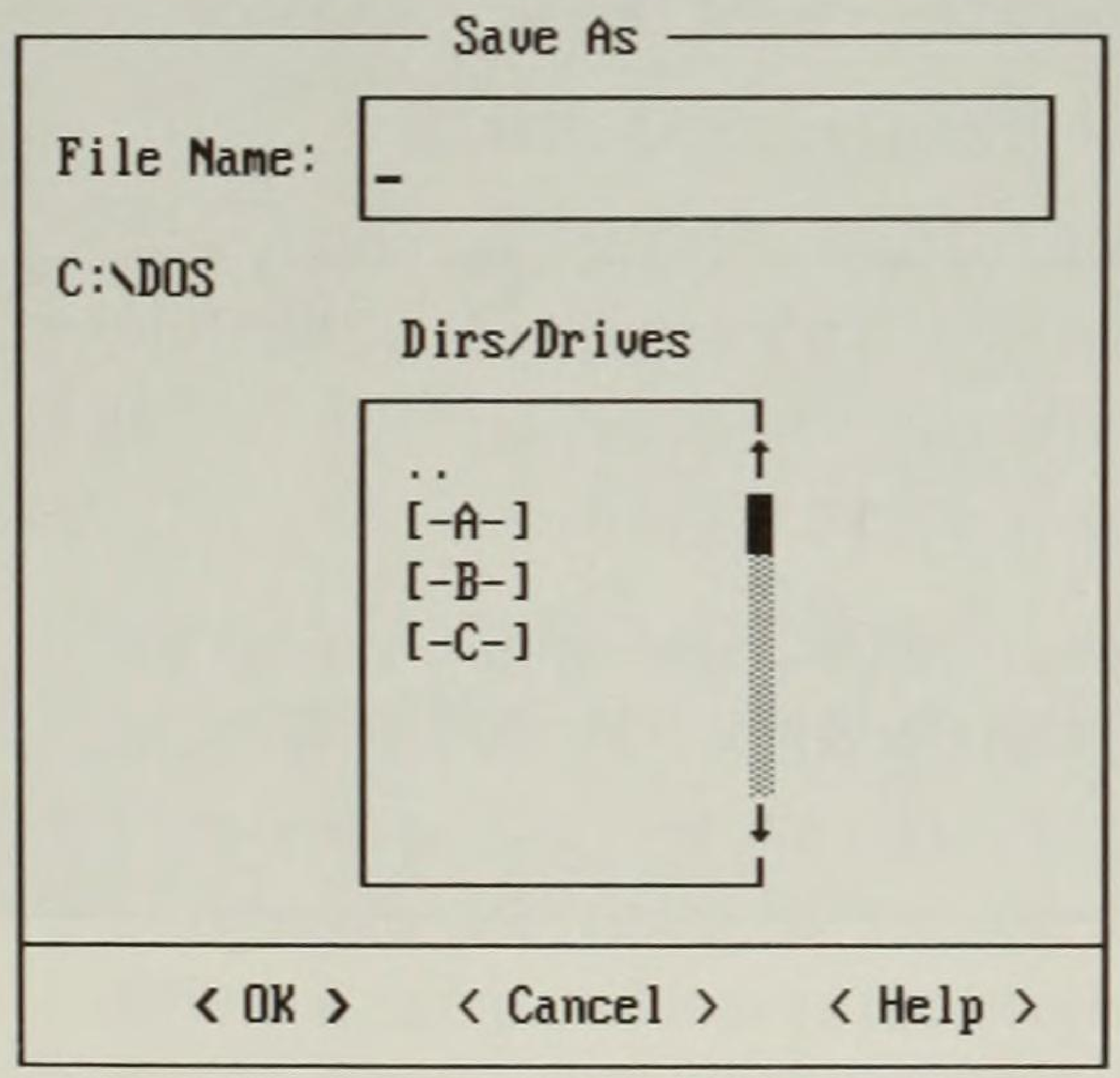
After you've typed in a program, especially one you want to keep, you should save it on disk. If you don't, and you turn off your computer, the program will be lost and you'll have to type it in again from scratch.

The program you just typed in isn't long and would require little effort to retype. But when your programs get longer, you'll definitely want to save them on disk because retyping would be unnecessary, extra work.

Practice: Saving a program on disk

1. Choose the Save As command from the File menu.

The Save As command displays the following dialog box:



2. Type the following into the File Name text box:

   ```
   my-first.bas
   ```

   NOTE: Because QBasic assumes that the file you are saving is a program that warrants a .BAS extension, it will put the .BAS extension on your program filename if you don't do it yourself.

3. Press Enter to carry out the Save As command. QBasic will save your program on disk under the filename MY-FIRST.BAS.

After you've saved the program on disk, QBasic returns you to the View window, which now displays the filename at the top of the window. This lets you know at all times what program you're currently working with. If it says *Untitled*, you know that you haven't saved the program on disk yet.

A program doesn't have to be complete before you save it. So how do you know when to save it for the first time, and how often after that do you need to save it again? Simply ask yourself how much work you'd lose and how upset you'd be if the power went off *right now*, and you should be able to answer that question.

Naming Files

The following hints will help you name your files effectively:

- A filename can be no more than eight characters long.
- The more descriptive the filename, the better.
- A filename can be a combination of letters, numbers, and some punctuation marks; you can't use spaces or the following characters:

  ```
  * = + [ ] ; : " ? < > / \ ¦
  ```

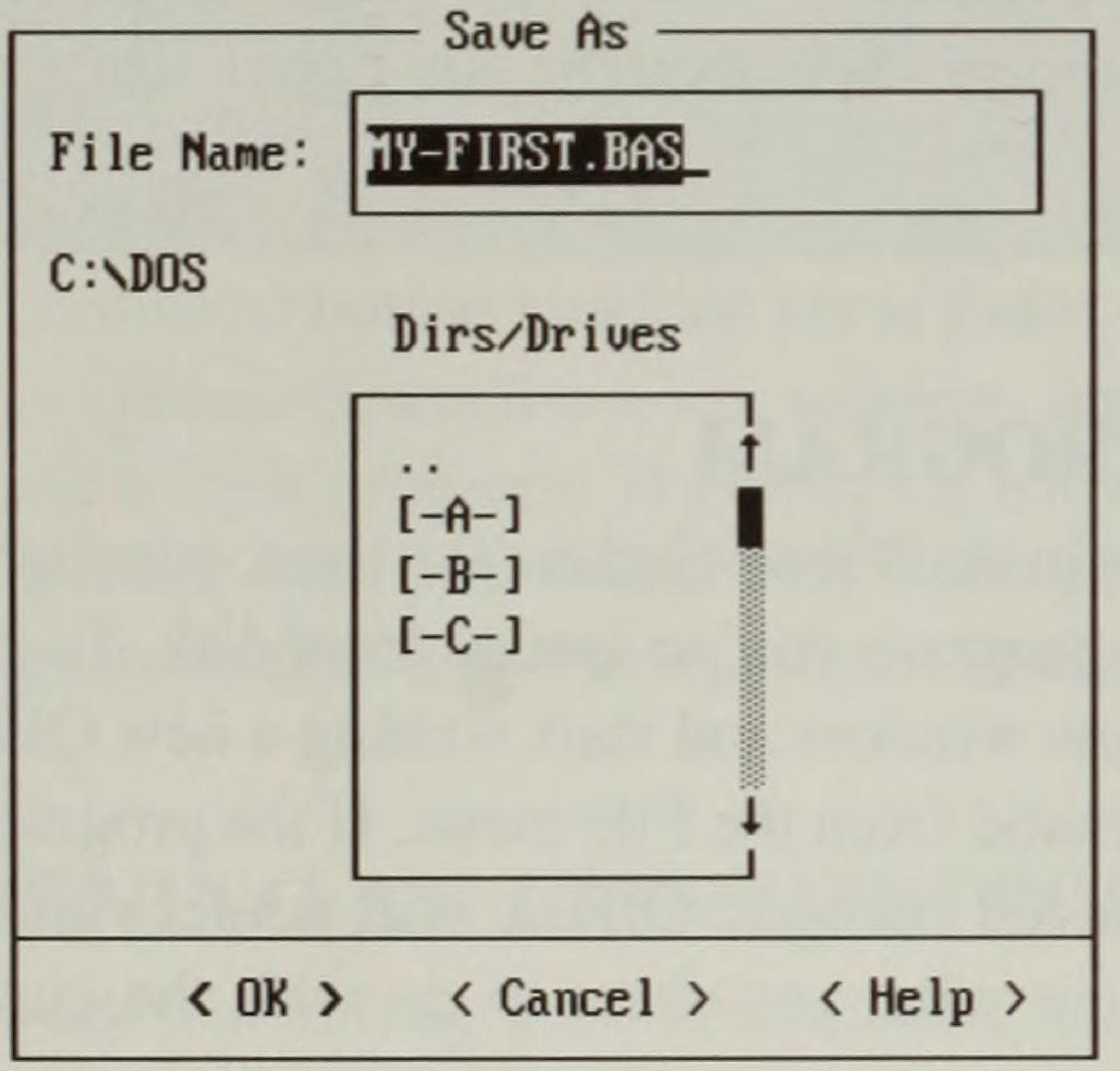
- The extension of your program filename should always be `.BAS`. This tells QBasic that the file is a legitimate QBasic program file.

Changing a Saved File

If, after using Save As to save a program on disk, you make any changes to the program, QBasic will treat the program in the View window as a separate program from the one you just stored on disk. QBasic will not let you work on another program until it knows what you want to do with the modified version of the current program: Do you want to overwrite the *saved* version of the file with the *changed* version of the file? Or do you want to discard the changes altogether?

Practice: Working with changes to a saved file

1. Make some changes to the program in the View window. The changes are up to you—maybe change the PRINT message a little or add a second PRINT instruction.
2. After making the changes, choose Save As from the File menu.



File Name: MY-FIRST.BAS

C:\DOS

Note that the File Name text box already contains the name you gave the program. You have several options at this point:

- ☐ Choose OK to save the changed version of the program on disk, replacing the original version.
- ☐ Type a new name for the changed program and choose OK. This saves the changed version of the program under a different name and leaves the original version intact.

- ☐ Choose Cancel to cancel the Save As operation and return to the View window.
- ☐ Choose Help to display another dialog box that explains the Save As dialog box.

3. Press Enter to choose OK and save the changed program under the same name on disk.

Save Often

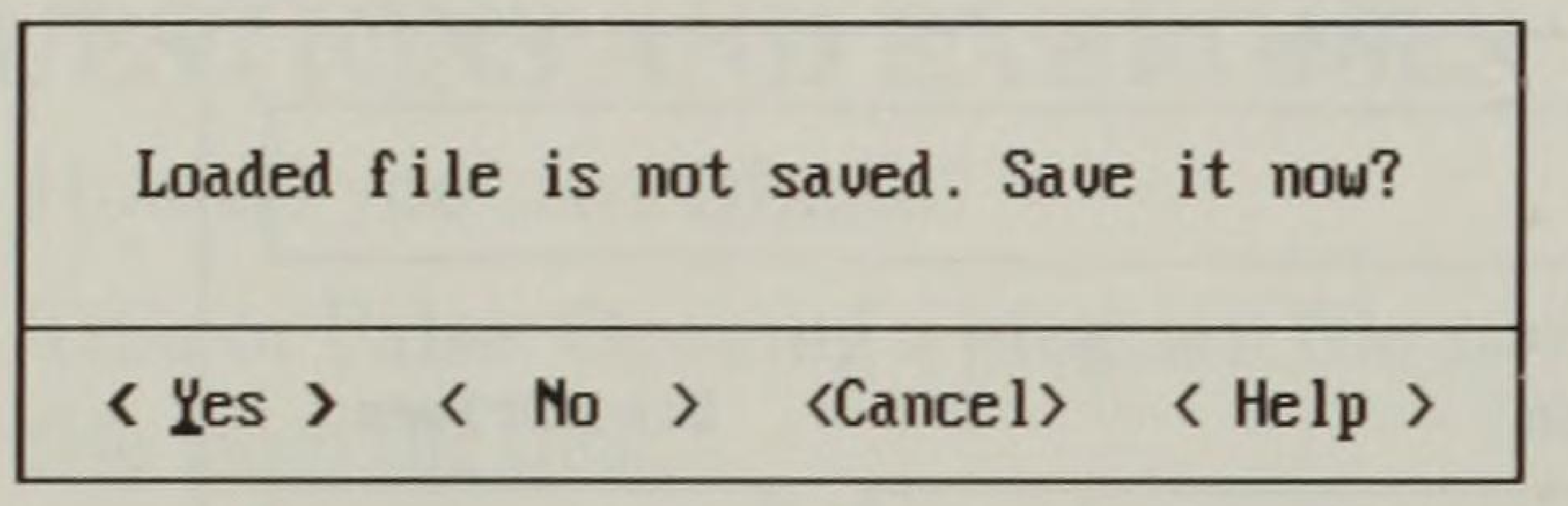
Until you save the program on disk, anything you type into the View window is held in the computer's temporary memory. Save your programs on disk regularly to avoid losing work. If your computer lost power as the result of a blackout or the accidental unplugging of the power cord, everything in the computer's memory—including your program—would be lost forever. If you had stored that program on disk, though, you could simply load it again.

STARTING A NEW PROGRAM

You can have only one program in the View window at a time, whether it's a program you've typed in or a program you've loaded from disk. To clear a current program from the View window and start writing a new QBasic program, choose the New command from the File menu. If the program in the View window contains unsaved changes, QBasic will not let you start a new program until it finds out what you want to do with the current changed program.

Practice: Starting a new program

1. Make another change to the program in the View window, such as changing the message PRINT displays.
2. After you've made the change, choose New from the File menu. The following dialog box appears.



Loaded file is not saved. Save it now?

If you press Enter to choose the default response *Yes*, QBasic saves the changed version of the program on disk using the name currently displayed at the top of the View window. This overwrites the older version already on the disk. You are not given the option of typing in another filename—to do that, you must first press Esc to exit from this dialog box, and then choose Save As from the Edit menu to give the changed program another name.

If you choose *No*, QBasic doesn't save the changes you've made since the last time you saved the file. Don't be too hasty about choosing *No*—be sure you're not discarding something you meant to save!

3. If you want to save your changed program and overwrite the older version, press Enter. Otherwise, press Tab once to highlight the No command button and then press Enter.

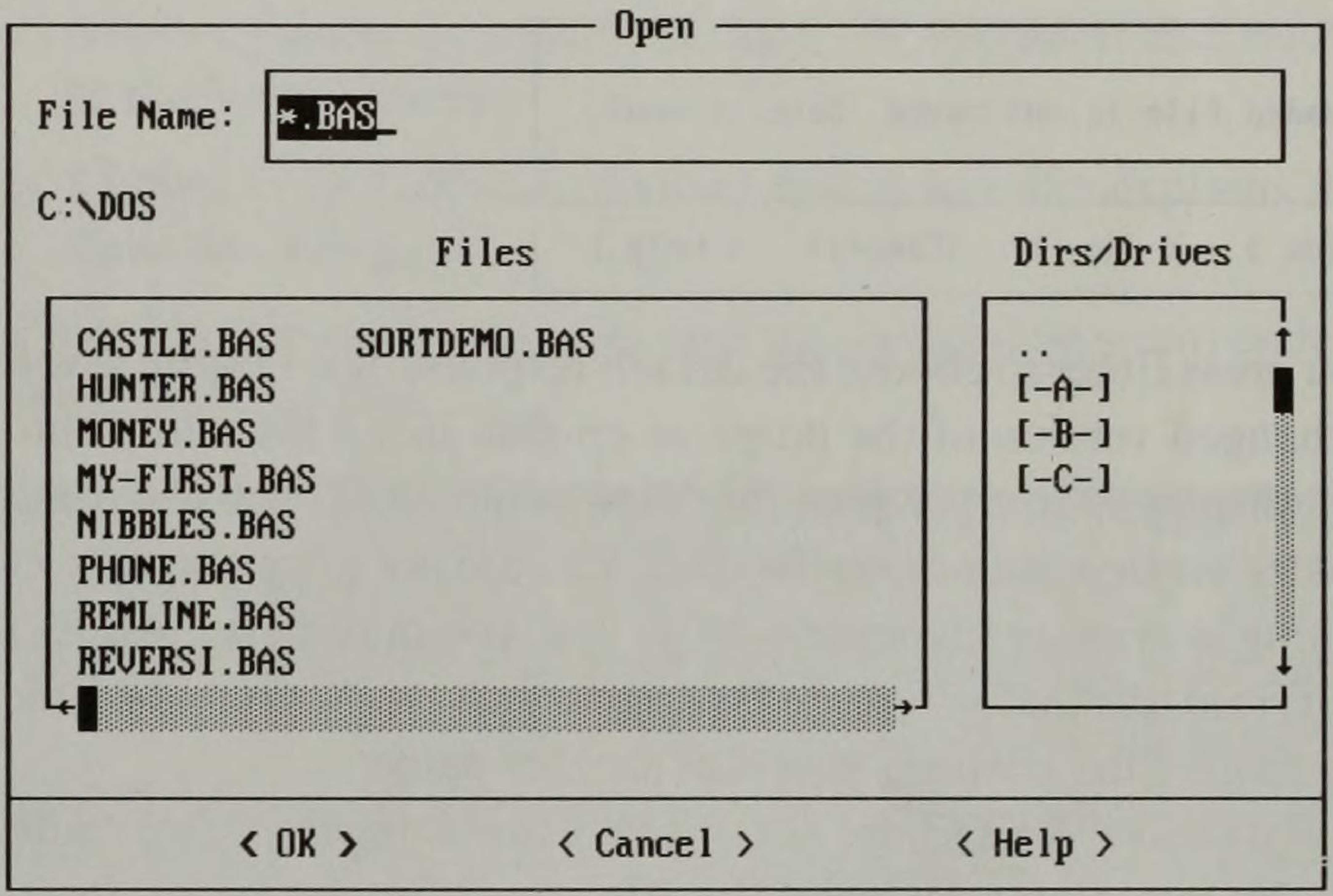
   QBasic clears the View window, and you're ready to start typing a fresh program. Try entering a few PRINT statements now to practice what you've learned. When you've finished, save the program on disk under an appropriate name.

OPENING AN EXISTING PROGRAM

To work with a program stored on disk, you must first open it, just as you must open a book before you can read it. Opening a program loads it into the View window. To open a program, you select the Open command from the File menu.

Practice: Opening an existing program

1. Drop down the File menu. Notice that the Open command is followed by an ellipsis (...).
2. Choose the Open command to display the Open dialog box.



```
Open
File Name: *.BAS
C:\DOS
Files                                   Dirs/Drives
CASTLE.BAS    SORTDEMO.BAS              ..
HUNTER.BAS                              [-A-]
MONEY.BAS                               [-B-]
MY-FIRST.BAS                            [-C-]
NIBBLES.BAS
PHONE.BAS
REMLINE.BAS
REVERSI.BAS
      < OK >        < Cancel >        < Help >
```

3. Press the Tab key to move the cursor to the Files list box.
4. Highlight MY-FIRST.BAS.
5. Choose OK. This causes QBasic to open the program and load it into the View window.

Now you are in a position to edit or run the QBasic program MY-FIRST.BAS. Use this method to load any existing file on disk. Use the Dirs/Drives list box to find programs elsewhere on disk.

QUITTING QBASIC

When you've finished programming, you should always exit QBasic before you turn off your computer. If you don't, the instant your screen goes dark you might realize that you forgot to save your program.

To quit QBasic, choose Exit from the File menu. Remember, if you have a program in the View window that you have not saved, or a program that you saved earlier but then changed, QBasic will ask if you intend to save the program.

SUMMARY

You're well on your way to learning to program in QBasic. You know how to start, use, and quit QBasic, and how to write, run, edit, and save your own programs. In the next chapter, you'll learn more QBasic instructions, and you'll look at QBasic instructions in a little greater detail.

QUESTIONS AND EXERCISES

1. How do you start QBasic?
2. True or False: Opening a program file stored on a disk erases that file from the disk.
3. What is the difference between the View window and the Immediate window?
4. On the File menu, what is the difference between the New command and the Open command?
5. What is the difference between the View window and the output screen?
6. On the Edit menu, what is the difference between the Cut command and the Copy command?
7. What are the four ways to delete text from a program, and how do they differ?
8. What is the difference between the insert cursor and the overstrike cursor? How can you tell the difference between the two? How do you change back and forth between the two?
9. Not including the .BAS extension, what is the maximum number of characters you can use to name your file?
10. Which of the following names *can't* you use to name your programs? Why not?

    PRINTNAME.BAS
    HELLO!.BAS
    BIG+BAD.BAS
    BIG RED.BAS
    HI-AND-LO.BAS
    BUSINESS.BAS
    BALLOON.BAS
    PROG[1].BAS
    50%DONE.BAS
    1ST*PROG.BAS
    MY_WAY.BAS
    IT'S-OK.BAS

11. How do you quit QBasic? What happens if you try to quit before you've saved your work, and what are your options?

CHAPTER 3

Introduction to the QBasic Language

In the last chapter you became familiar with QBasic, and you used a couple of QBasic instructions to write a short program. In this chapter you'll take a closer look at the QBasic language and expand upon what you've already learned.

ANATOMY OF A QBASIC INSTRUCTION

So far, we've referred to the QBasic lines you've been typing in as *instructions.* As you learned in Chapter 1, each line in a QBasic program is simply an instruction that the computer carries out when you run the QBasic program. QBasic recognizes two types of instructions: *statements* and *functions.*

Statements and Functions

Statements and functions look very much alike and are equally easy to use. They differ primarily in purpose:

- A statement is usually straightforward. It simply does what you tell it to do when you run the program. When your program executes a statement, the result is usually apparent. The PRINT statement, for example, puts characters on the output screen, and the CLS statement clears the output screen.
- A function returns a value that your program can use. A function usually works in a less obvious way than a statement does. A function appears within a statement and performs its work when the statement is executed.

Each statement and function has a *syntax,* which dictates how the statement or function must be used in your program.

QBasic Syntax

The syntax of a QBasic instruction, whether statement or function, is simply a keyword followed by the information that the instruction needs to do its work.

The syntax of some QBasic instructions, such as BEEP, consists of only the name of the statement or function. Start QBasic (if you haven't already done so), and type in the following single-line program:

```
BEEP
```

Run the program and see what happens. The BEEP statement causes the speaker inside your computer to emit a brief tone.

Now let's take a look at some instructions of greater substance. Throughout this chapter we'll use the PRINT statement as our example.

PRINT statement syntax

The PRINT statement has the following syntax:

```
PRINT [expressionlist][,¦;]
```

Looks a bit different from the PRINT statement in Chapter 2, doesn't it? The PRINT statement is quite versatile; in the previous chapter, you used only one of the PRINT statement's many features.

Let's take a moment to become familiar with the structure of a syntax line (Figure 3-1):

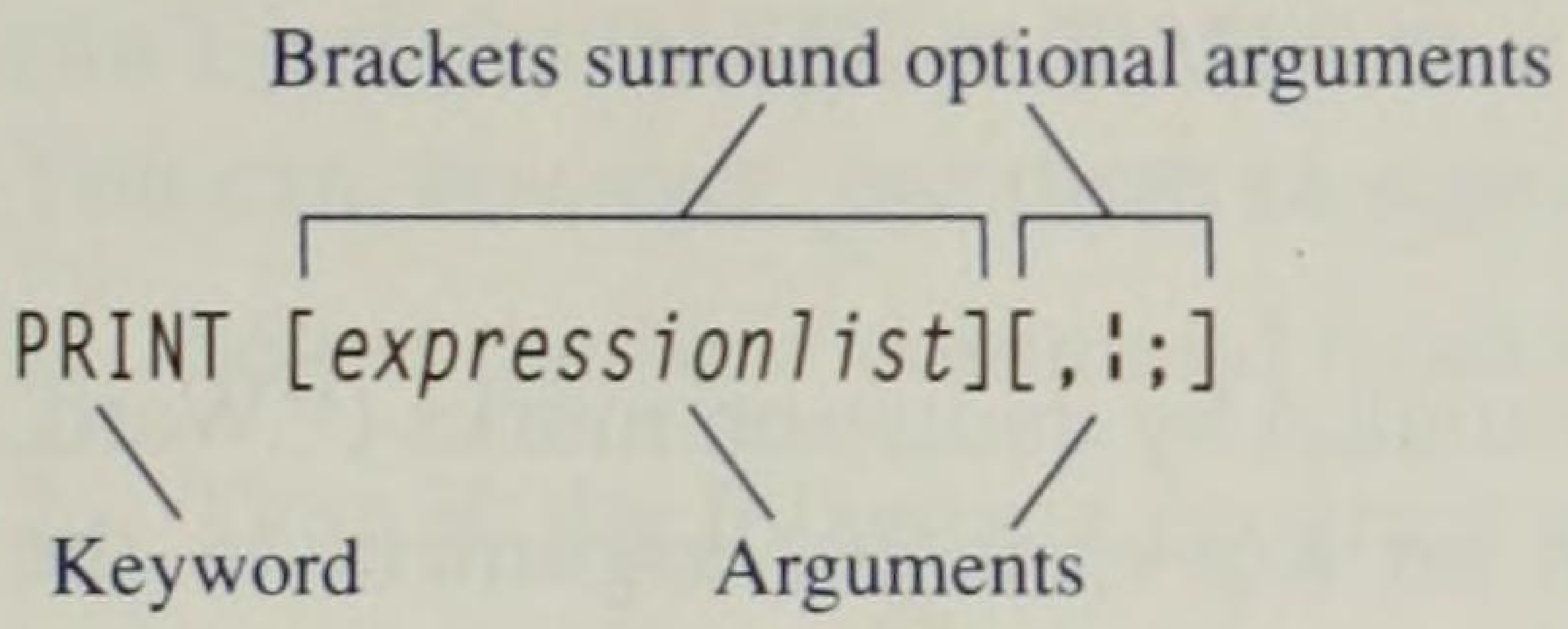

Brackets surround optional arguments
PRINT [expressionlist][,¦;]
Keyword
Arguments

FIGURE 3-1.
Structure of a syntax line.

Each item in a syntax line except for the keyword is called an *argument.* Arguments provide the instruction with the additional information it needs in order to do its work. Some arguments are optional; they appear within square brackets both in this book and in QBasic's built-in help system. If you decide to use one of these optional arguments in your instruction, do not type in the square brackets. (Along the same lines, do not type in the ¦ symbol—it simply indicates that you are to choose one of the arguments surrounding the ¦. In the case of Figure 3-1, you would choose either the comma or the semicolon.)

In a PRINT statement, both arguments are optional: You can use the PRINT keyword by itself.

Practice: Using a PRINT statement without arguments

1. Select the New command from the File menu.
2. Type in the following program and run it:

```
CLS
PRINT "This statement uses an argument."
PRINT
PRINT "So does this one."
```

Your screen will look like this:

```
This statement uses an argument.

So does this one.
```

As you can see, a PRINT statement without any arguments prints a blank line. (Simply pressing Enter to create blank lines won't work.) Let's look at three types of arguments you can use with a PRINT statement: text, numeric expressions, and functions.

Using text as an argument

In the last chapter, you used text surrounded by quotation marks (''Ward, I'm worried about the Beaver.'') as an *expressionlist* argument to the PRINT statement. Such text is called a *string*. You can use any numbers, letters, spaces, and punctuation marks—except double quotation marks—in a string. (QBasic perceives a double quotation mark within a string as the end of the string.)

Using numeric expressions as arguments

Another valid *expressionlist* argument is a *numeric expression*. A numeric expression is a number, a mathematical equation, or, as you'll learn in the next chapter, a special kind of word called a numeric *variable*. Numeric expressions are not enclosed in double quotation marks.

Practice: Using numeric expressions as arguments

1. Modify your program so that it contains only the following PRINT statement:

```
CLS
PRINT 42
```

2. Now run the program. Your output screen will look like this:

```
 42
```

Using functions as arguments

A third valid *expressionlist* argument is a QBasic *function.* A function generally performs a background task and returns information you can use in your program. A function is not enclosed in double quotation marks.

Practice: Using functions as arguments

QBasic has two functions, named DATE$ and TIME$, that obtain the current date and time from the clock that is built into your computer. You cannot use these functions by themselves. (In other words, you can't just put DATE$ or TIME$ on a line and expect anything to happen.) You can, however, use them as arguments for a PRINT statement.

1. Select the New command from the File menu.
2. Type in the following program and run it:

```
CLS
PRINT DATE$
PRINT TIME$
```

Your output screen will look similar to this:

```
03-25-1991
21:13:09
```

Although the date and time displayed here don't match what you see on your output screen, the underlying principle remains sound: DATE$ and TIME$ performed a task and returned a value in an unobtrusive manner. That's the beauty of QBasic functions: You simply use them, and they perform useful work without your having to worry about the details.

Printing More than One Item with PRINT

So far, you've used a single argument, such as a single string or a single function, as an argument for each of your PRINT statements. You can use multiple arguments in a single PRINT statement, but only if you use a special character called a *separator* between them. The PRINT statement recognizes two separator characters: a comma and a semicolon.

The comma separator

You can think of the comma separator as the programming equivalent of the Tab key. When PRINT encounters a comma, it prints the value of the next argument at the beginning of the next print zone. (Print zones are the programming equivalent of tab stops; they are 14 characters long.) This ability to print values at specific locations makes comma separators a natural choice when you want to print information in columns.

Practice: Using comma separators

1. Select the New command from the File menu.
2. Type in and run the following program to see how comma separators work:

```
CLS
PRINT "Comma", "separators", "separate", "arguments"
PRINT "They", "also", "align", "arguments"
```

Your output screen will look like this:

```
Comma         separators    separate      arguments
They          also          align         arguments
```

NOTE: If you did not put a space after each comma, QBasic put them in for you when you pressed Enter to end the line. QBasic always adds a space after a separator if you forget, giving your programs a neat, consistent appearance.

You can even use commas by themselves without having an argument between them, as shown in the following practice session.

Practice: Positioning arguments with comma separators

1. Select the New command from the File menu.
2. Type in and run the following program, which uses commas to position values near the center of the screen:

```
CLS
PRINT , , "Use comma separators"
PRINT , , "to position arguments"
```

Your output screen will look like this:

```
                            Use comma separators
                            to position arguments
```

The semicolon separator

When PRINT encounters a semicolon separator, it prints the next argument immediately after the argument it just printed. PRINT places no spaces between these arguments: You must supply them if you want them.

Practice: Using semicolon separators

1. Select the New command from the File menu.
2. Type in and run the following program, which uses semicolon separators with and without added spaces:

```
CLS
PRINT "This"; "is"; "what"; "semicolons"; "do"
PRINT "If "; "you "; "want "; "spaces, "; "add "; "them"
```

Your output screen will look like this:

```
Thisiswhatsemicolonsdo
If you want spaces, add them
```

Using both comma and semicolon separators

Just as you can use more than one kind of argument on a line, you can use more than one kind of separator on a line. You can mix commas and semicolons however you like, and they always work as advertised.

Practice: Using both comma and semicolon separators

1. Select the New command from the File menu.
2. Type in and run the following program, which uses commas and semicolons on the same line:

   ```
   CLS
   PRINT "These"; "words"; "run"; "together", "These", "are", "apart"
   ```

   Your output screen will look like this:

   ```
   Thesewordsruntogether       These         are           apart
   ```

Using a separator at the end of a PRINT statement

When you place a comma or semicolon separator at the end of a PRINT statement, you dictate where the result of the *subsequent* PRINT statement appears on the output screen:

- A comma prints the next PRINT statement's output at the beginning of the next print zone.
- A semicolon prints the next PRINT statement's output immediately following the output from the current PRINT statement.

Practice: Using separators at the end of a PRINT statement

1. Select the New command from the File menu.
2. Type in and run the following program, which uses a semicolon at the end of a PRINT statement:

   ```
   CLS
   PRINT "This is the first line ";
   PRINT "and this is the second"
   ```

   Your output screen will look like this:

   ```
   This is the first line and this is the second
   ```

3. Now change the semicolon to a comma, and run the program again.

   NOTE: If you use the semicolon at the end of a PRINT statement and there are no more PRINT statements in your program, it has no effect.

SUMMARY

In this chapter you learned more about QBasic program instructions and learned how various syntax options make a QBasic statement more versatile. In the next chapter, you'll learn about two important QBasic elements that will add even more flexibility to your programs: variables and operators.

QUESTIONS AND EXERCISES

1. Briefly describe the difference between a statement and a function.
2. Which of the following QBasic keywords are statements and which are functions?

   BEEP
   DATE$
   TIME$
   CLS
   PRINT

3. In a QBasic syntax line, what is the significance of square brackets around an item? What does the ¦ character signify?
4. What is an argument?
5. What is the difference between a string and a numeric expression?
6. How will the output from the following two PRINT statements differ?

   ```
   PRINT "Hello", ; "there"
   PRINT "Hello"; , "there"
   ```

7. What happens when you put a semicolon or a comma at the end of a PRINT statement?

CHAPTER 4

QBasic Variables and Operators

MOD
OR
OR
MOD
AND

As you program, you may need to print certain numbers or strings of characters more than once. QBasic provides you with a handy method of doing just that—a method that doesn't require a lot of retyping on your part. In QBasic, you can store data and use it whenever and as often as you like, simply by using the name of the storage location in your program. You name the storage location, which is known as a *variable,* according to the type and size of data it contains.

WHAT DO YOU WANT TO STORE?

Variables are of two types: *string* and *numeric.* A string variable is a name representing a storage location for a string. A numeric variable is a name representing a storage location for a number. In addition, numeric variables have several subtypes. Figure 4-1 shows the valid variable types in QBasic.

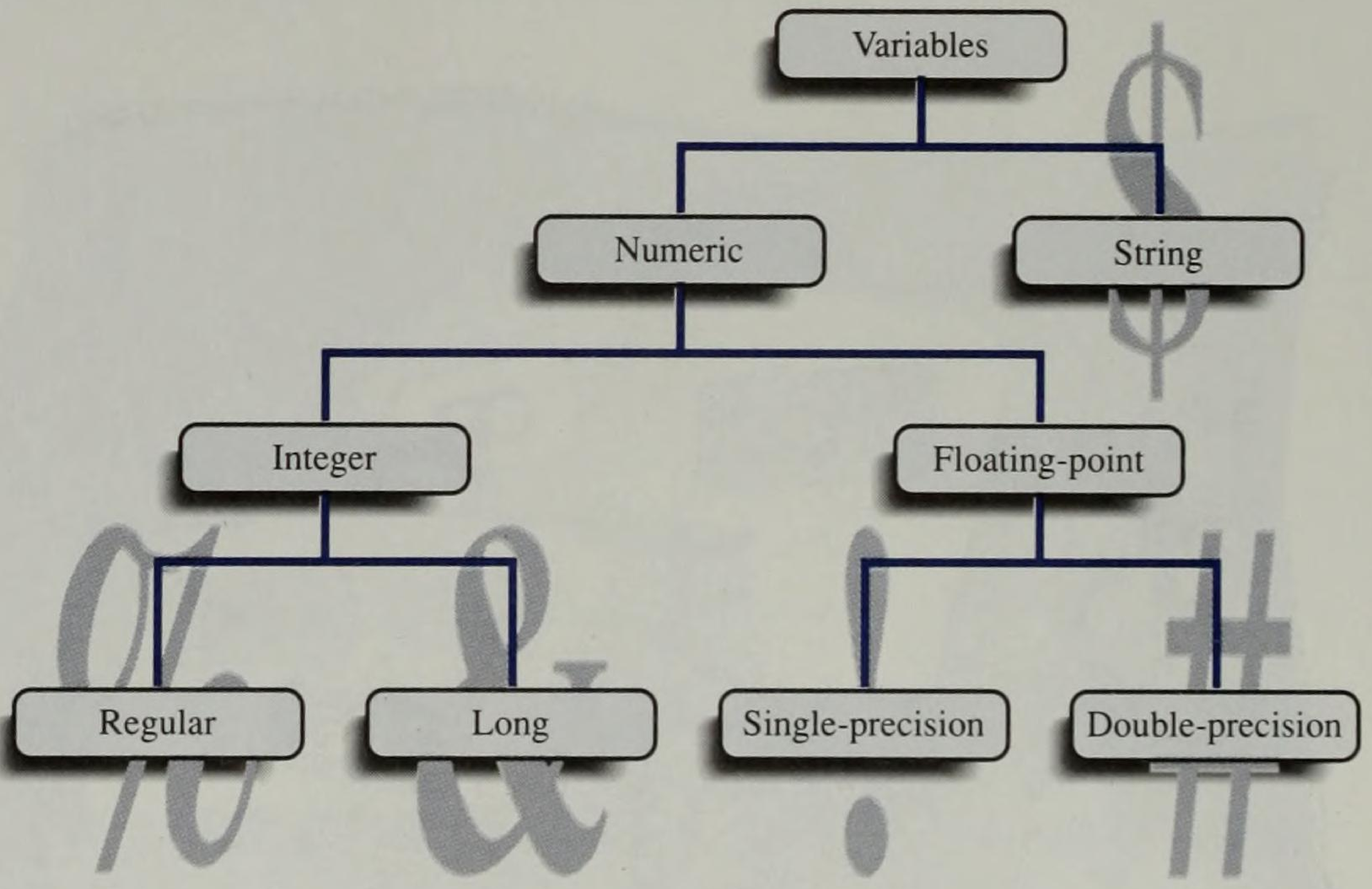

Variables
Numeric
String
$
Integer
Floating-point
Regular
Long
Single-precision
Double-precision
%
&
!
#

FIGURE 4-1.
An overview of variable types used in QBasic.

As you read through this chapter, keep the following questions in mind. As you'll see, the answers to these questions help you determine which type of variable is most suited to your purposes.

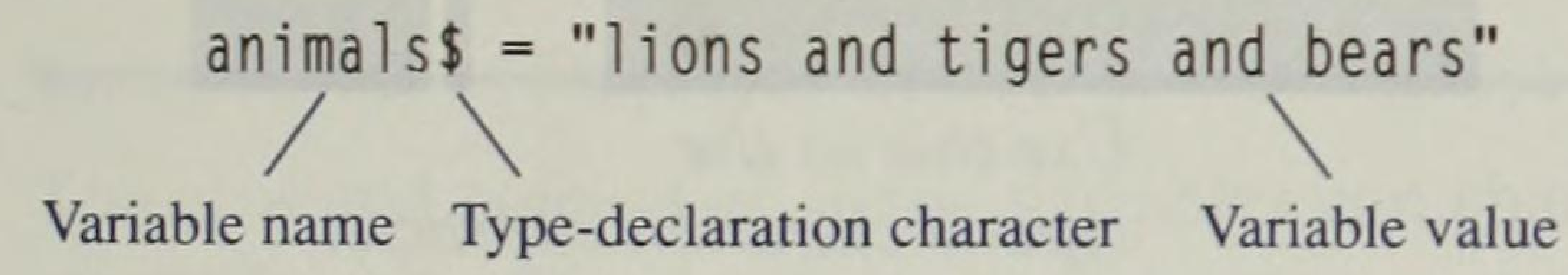
- **Is your data text or a number?** Text is stored in a string variable; a number is stored in a numeric variable.
- **If you are storing a number, does it have a decimal point?** An integer—that is, a number without a decimal point—is stored in an *integer variable.* A number with a decimal point is stored in a *floating-point variable.*
- **How large is the number you want to store?** The size of the number plays a part when you determine which variable to use.

USING VARIABLES—AN OVERVIEW

When you decide to use a variable, you must first *declare* it in your program; that is, you must inform QBasic of three things:

- The variable's name
- The variable's type
- The variable's value

You can see the elements of a declaration in Figure 4-2, which shows a string variable declaration as an example.



```
animals$ = "lions and tigers and bears"
```

FIGURE 4-2.
The elements of a variable declaration.

The following sections describe how to use each of these elements.

Naming a Variable

These rules and suggestions will help you choose appropriate names for your QBasic variables:

- A variable name can be as many as 40 characters in length.

- A name can be any combination of uppercase and lowercase letters. (Keep in mind, however, that all-uppercase names might be hard to distinguish from QBasic keywords.)
- You can't use QBasic keywords, such as PRINT or BEEP, as variable names.
- The appropriate type-declaration character must be the last character of the variable name. (See "Declaring the Variable Type.")
- The most descriptive variable names are usually the most effective. For example, the string variable names *firstName$* and *lastName$* leave little doubt in your mind as to what these variable names represent.

NOTE: QBasic is sensitive to how you use uppercase and lowercase letters. So sensitive, in fact, that if you type in a variable name you've already used with different capitalization, QBasic adjusts the previously typed names to reflect the new capitalization.

Declaring the Variable Type

If you look again at Figure 4-2, you'll notice that the final character of the variable name is the *type-declaration character*. The type-declaration character tells QBasic what type of variable you are declaring. Each type has its own symbol:

If the variable is	***Use this as the type-declaration character***
String	$
Integer	
Regular	%
Long	&
Floating-point	
Single-precision	!
Double-precision	#

When QBasic encounters the type-declaration character in the variable name, it knows what kind of data is stored in that variable.

Declaring the Value of a Variable

When you declare the value of a variable, be sure that the value type matches the type-declaration character. For example, an integer variable can't hold a string value.

What Next?

Now that you've been introduced to the overall structure of a variable declaration, you can begin to learn about the different types of variables themselves. First you'll learn about string variables and numeric variables. Then you'll learn about getting information from the keyboard and storing it in variables. The final part of this chapter describes how QBasic works with mathematics in general.

STRING VARIABLES IN QBASIC

A string variable name represents a storage location for a string of text that you want to use throughout a program.

To declare a string variable, use the dollar sign type-declaration character ($) at the end of the variable name. Enclose the string in double quotation marks exactly as you would in a PRINT statement. For example,

```
beatAuthors$ = "Kerouac, Ginsberg, Ferlinghetti"
```

The size and content of string variables can change within a program, as you'll learn in Chapter 9.

Practice: Declaring and using a string variable

1. Select the New command from the File menu.
2. Type in and run the following program, which assigns a value to a string variable and then prints the string:

   ```
   CLS
   tvDocs$ = "Welby, Casey, Kildare, McCoy"
   PRINT tvDocs$
   ```

Your output screen should look like this:

```
Welby, Casey, Kildare, McCoy
```

NUMERIC VARIABLES IN QBASIC

QBasic uses two types of numeric variables: *integer numeric variables* and *floating-point numeric variables.* Each type has subtypes designed for numbers of different sizes. (Remember, we told you earlier that the size of the number helps you determine which type of variable to use.) Figure 4-3 shows the relationships among numeric variable types in QBasic.

First you'll learn about the two types of integer variables and when to use them; then you'll be introduced to the two types of floating-point variables.

Integer Variables

QBasic uses two types of integer variables: *regular integer variables* and *long integer variables.* The difference between the two lies in the size of the integer number each can represent.

How QBasic Looks at Numbers

Whether you're balancing your checkbook or preparing reports and proposals, you're likely to spend a little time each day working with numbers. What you probably don't realize is that you are actually working with different *types* of numbers. For a lot of people, the ''proper'' names for these numbers—whole numbers, integers, and so on—are only a dim memory from a grade-school math class. But to QBasic, the differences between two particular types of numbers are crucial:

- An integer is a number with no decimal point.
- A floating-point number is a number with a decimal point.

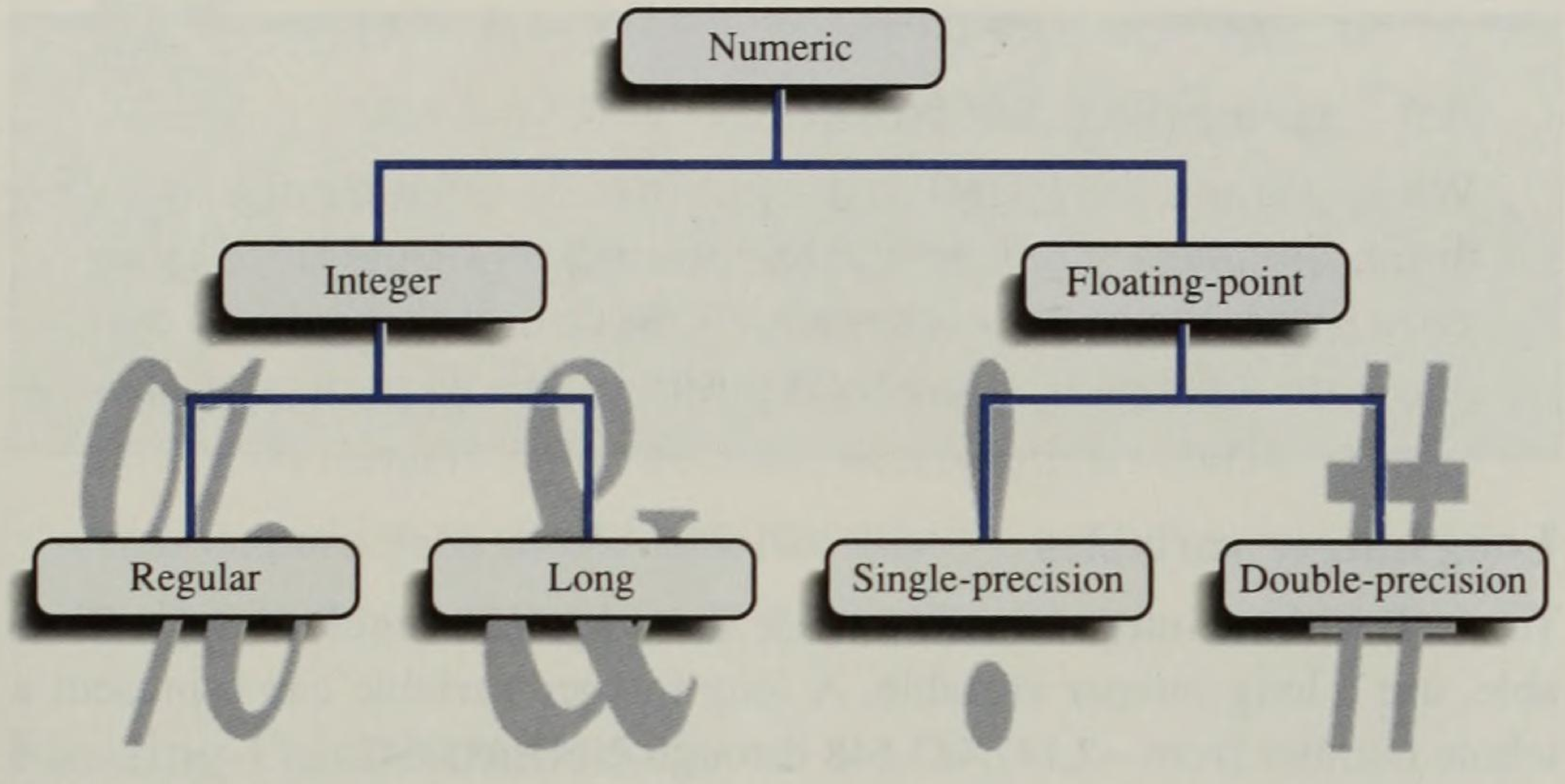

Numeric
Integer
Floating-point
Regular
Long
Single-precision
Double-precision
%
&
!
#

FIGURE 4-3.
Numeric variables in QBasic.

Regular integer variables

A regular integer variable can store any whole number from –32,768 through 32,767.

To declare a regular integer variable, use the percentage sign type-declaration character (%) at the end of the variable name. For example,

```
henrysWives% = 6
```

Practice: Declaring and using a regular integer variable

1. Select the New command from the File menu.
2. Type in and run the following program, which declares a regular integer variable and prints its value:

   ```
   CLS
   daysofXmas% = 12
   PRINT daysofXmas%
   ```

   Your output screen should look like this:

   ```
    12
   ```

An Extra Space for Numbers

When you use the PRINT statement in your QBasic program to display numbers, you'll notice that all numbers are preceded by an extra space. If a number is negative, QBasic uses this space to display a minus sign. If a number is positive, QBasic prints a space.

Long integer variables

To work with an integer that is outside the range of a regular integer variable, use a long integer variable. A long integer variable can represent a whole number from –2,147,483,648 through 2,147,483,647.

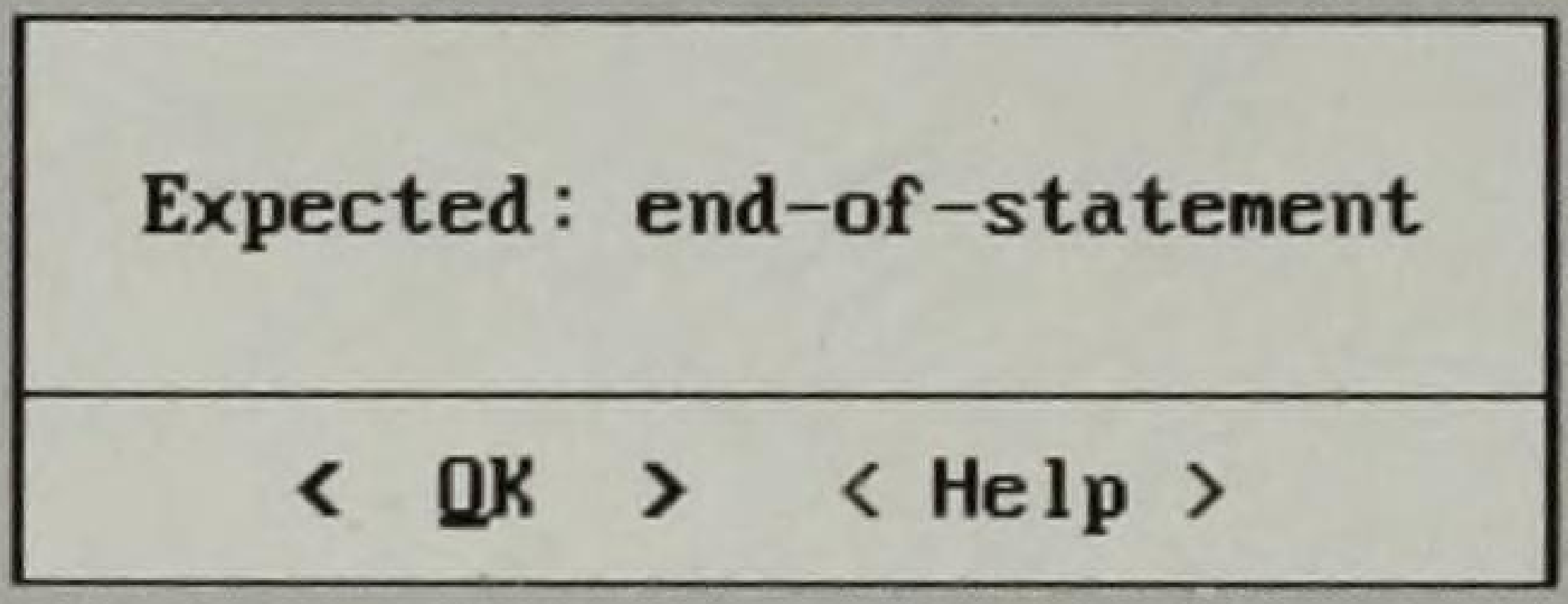
To declare a long integer variable, use the ampersand type-declaration character (&) at the end of the variable name. For example,

```
cityPopulation& = 175000
```

You Can't Use Commas in Numbers

In everyday life, you use commas to break large numbers into recognizable segments. (For example, the number 1,653,892,000 is certainly easier to comprehend than 1653892000.) But because commas have a special use as separator characters in QBasic, you must learn to get along without them when you enter numbers.

If you forget and try to use a comma as part of a number within your program, QBasic displays an error message:



```
Expected: end-of-statement
```

Likewise, if someone running your program types in a number with commas, QBasic asks the user to type the number in again.

Practice: Declaring and using a long integer variable

1. Select the New command from the File menu.
2. Type in and run the following program, which declares a long integer variable and prints its value:

```
CLS
freefall& = 84700
PRINT "The longest free fall, in feet, was"; freefall&
```

Your output screen should look like this:

```
The longest free fall, in feet, was 84700
```

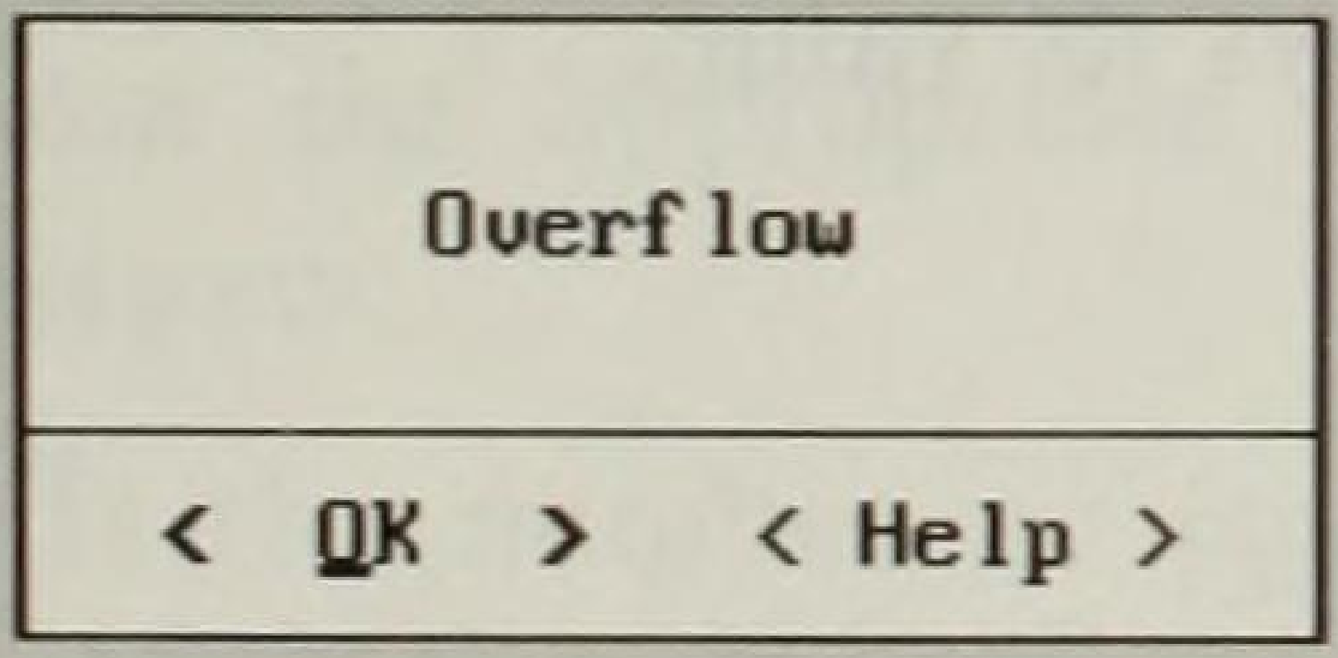
Floating-Point Numbers

So far you've worked only with integers. In real life, however, you often deal with numbers that have a fractional part—that is, a number followed by a decimal point and the decimal expression of a fraction.

QBasic provides two variable types to represent floating-point numbers: *single-precision floating-point variables* and *double-precision floating-point variables.* The primary difference between the two variable types lies in how accurately they can represent a given floating-point number.

Using a Variable That's the Wrong Size

As you've seen, working with numbers in QBasic requires a lot of attention to the size of a number. If you accidentally assign an out-of-range number to a variable, QBasic displays a dialog box that indicates that the number is invalid for that variable type:



Overflow

You can help prevent such mistakes by choosing variables of the correct size and displaying the range of valid values for the user.

- A single-precision floating-point variable can represent a number up through 7 digits in length. The decimal point can be anywhere within those digits.
- A double-precision floating-point variable can represent a number up through 15 digits in length. The decimal point can be anywhere within those digits.

The following table shows some sample floating-point values:

Single-precision	*Double-precision*
412.002	1.000032478
19246.34	4280000.0055
.00025	7160.0000000005
657926.3	.10000000001

Single-precision floating-point variables

To declare a single-precision floating-point variable, use the exclamation point type-declaration character (!) at the end of the variable name. For example,

```
lpSpeed! = 33.3333
```

Practice: Declaring and using a single-precision floating-point variable

1. Select the New command from the File menu.
2. Type in and run the following program, which declares a single-precision floating-point variable and displays its value.

```
CLS
carPrice! = 12999.99
PRINT "This sporty new car costs"; carPrice!
```

Your output screen should look like this:

```
This sporty new car costs 12999.99
```

Double-precision floating-point variables

Double-precision floating-point variables, which can be up through 15 digits in length, are useful for scientific work that requires precise numbers.

To declare a double-precision floating-point variable, use the pound sign type-declaration character (#) at the end of the variable name. For example,

```
bigFloat# = 5.0000000127
```

Practice: Declaring and using a double-precision floating-point variable

1. Select the New command from the File menu.
2. Type in and run the following program, which declares a double-precision floating-point variable and displays its value:

```
CLS
pi# = 3.141592653589793
PRINT "The value of pi is"; pi#
```

Your output screen should look like this:

```
The value of pi is 3.141592653589793
```

Which Numeric Variable Type Should You Use?

You've learned that QBasic provides four types of numeric variables for you to use in your programs: regular integer, long integer, single-precision floating-point, and double-precision floating-point. When you want to assign a numeric value to a variable name in your program, you have to choose the appropriate type of variable. Ask yourself the following questions:

- Is the number an integer? Will the number remain an integer throughout the program? If so, use an integer variable.
- Does the number have a decimal point? Or, more important, *might* the number have a fractional part later on in the program? If so, use a floating-point variable.

Once you've determined the type of variable—integer or floating-point—you have to determine what size variable to use.

Choosing the proper variable size

At first glance you might think that because they can hold large numbers, long integer variables and double-precision floating-point variables would be the obvious choices to use at all times. After all, using these larger-size variables would certainly reduce your chances of receiving *Overflow* error messages.

However (you knew there had to be a catch, right?), these larger variables have a disadvantage that you need to be aware of. Their size does provide you with greater storage space for your values, but they also take up a lot of memory in your computer. When you use the large variables unnecessarily, it's like using a grain silo to store a five-pound bag of flour. Computer memory, like acreage, is limited. If you build a silo every time you buy a bag of flour, you eventually use up your acreage. Take the time to make wise use of your resources; determine the smallest possible variable type you can use—then use it.

Don't use a larger variable than you need.

Changing the contents of a variable

The word "variable" itself is one clue to why variables are so useful: Their contents can vary depending on the needs of the program or the needs of the user. You can change the contents of a variable simply by assigning a new value to the variable. Remember, however, that the *current value* of a variable is the value it was last assigned.

Practice: Changing the value of a variable

1. Select the New command from the File menu.
2. Type in and run the following program, which declares and then changes the value of a variable:

```
CLS
fruit$ = "apple"
PRINT fruit$
fruit$ = "orange"
PRINT fruit$
fruit$ = "pear"
PRINT fruit$
```

Your output screen will display the following result:

```
apple
orange
pear
```

USING USER-SUPPLIED INFORMATION IN A VARIABLE

Now that you've become acquainted with variables, let's look at an area of QBasic programming that relies heavily on the use of variables, namely, getting characters from the person using a program. This process is sometimes called "reading characters from the keyboard."

Reading Characters with the INPUT Statement

The most commonly used QBasic statement for reading characters from the keyboard is the INPUT statement. You can't use an INPUT statement by itself, however. You must follow the INPUT statement with the variable name (including the variable type) you want to assign to the characters to be typed by the user.

Practice: Using the INPUT statement

1. Select the New command from the File menu.
2. Type in and run the following program:

```
CLS
PRINT "Type in a word and press Enter"
INPUT word$
PRINT "The word you typed in was "; word$
```

Your output screen will display the result of the operation:

```
Type in a word and press Enter
? Buckaroo
The word you typed in was Buckaroo
```

Take a look at what happened:

- First your program cleared the screen, and then it used a PRINT statement to tell the user what to do.
- Following the instructions of the PRINT statement, you typed in a word and pressed Enter. The word you typed in was assigned by the INPUT statement to the *word$* variable.
- The final PRINT statement used both a string and the *word$* variable to show what you typed in.

Asking the user for input

The preceding program used a PRINT statement to ask the user to supply information. You can, however, make the message requesting input a part of your INPUT statement. This forces the INPUT statement to do double duty: It both asks the user for input *and* reads in what the user types.

Practice: Placing a message within INPUT

Change the program you just typed in to this:

```
CLS
INPUT "Type in a word and press Enter"; word$
PRINT "The word you typed in was "; word$
```

After you run the program, your output screen will look like this:

```
Type in a word and press Enter? Buckaroo
The word you typed in was Buckaroo
```

Note the differences between this program and the one you typed in earlier:

- Because the user-friendly request is part of the INPUT statement, the cursor waits for your input on the same line as the request, instead of on a line by itself.
- The INPUT statement displays a question mark at the end of the message. (To eliminate the question mark, simply change the semicolon at the end of the INPUT message to a comma and rerun the program.)

Practice: Getting creative with INPUT

Now that you know a little about variables and how to get information from the keyboard using INPUT, you can add a little spice to your programs.

1. Select the New command from the File menu.
2. Type in and run the following program:

```
CLS
INPUT "Type in your first name and press Enter:  ", firstName$
PRINT
INPUT "How old are you (I promise not to tell anyone)?  ", age%
PRINT
INPUT "How much change do you have in your pocket?  $", money!
PRINT
PRINT "Thank you, "; firstName$; ".  You said that you are"; age%
PRINT "years old and have $"; money!; "worth of change."
```

After you run the program, your output screen will look similar to this:

```
Type in your first name and press Enter: Pat

How old are you (I promise not to tell anyone)? 45

How much change do you have in your pocket? $1.16

Thank you, Pat.  You said that you are 45
years old and have $ 1.16 worth of change.
```

QBASIC MATHEMATICAL OPERATORS

QBasic provides several symbols you can use to perform mathematical calculations in your programs. These symbols, called *operators,* let you perform tasks such as addition, subtraction, multiplication, and division.

Several QBasic operators are the same as those you use in everyday life. For example, in QBasic you use + for addition and – for subtraction. Other QBasic operators are represented by special symbols. The following table shows the operators you can use in QBasic:

Operator	*Mathematical operation*
+	Addition
–	Subtraction
*	Multiplication
/	Division
\	Integer division
MOD	Remainder division
^	Exponentiation (raising to a power)

The last three operators (\, MOD, and ^) are special-purpose operators that we'll describe shortly.

Commenting Your Programs

To help you keep track of what your program does, QBasic lets you place comments within your programs. Comments are preceded by the REM statement or the ' symbol and are for the programmer's use only. Comments do not appear in output produced by the program. Type in and run the following program to see how the REM statement works:

```
REM SAMPLE.BAS
REM A sample program demonstrating the use of comments.
REM
REM Programmers:  Mike and Dave
REM Date:  November 1, 1991

CLS

PRINT "This is a program with comments!"
```

The program produces the following output:

```
This is a program with comments!
```

The ' symbol can also be used as a less obtrusive equivalent of the REM statement, as shown in the following program:

```
' SAMPLE.BAS
' A sample program demonstrating the use of comments.
'
' Programmers:  Mike and Dave
' Date:  November 1, 1991

CLS

PRINT "This is a program with comments!"
```

When you run this program, it produces the same result:

```
This is a program with comments!
```

We'll be using the second commenting style in the programs in this book. As you write your own programs, be sure to include comments not only for yourself but also for others who might need to make sense of your program someday.

Working with QBasic Operators

To perform calculations in QBasic, simply use the operators as you would in "real life." For example, to add the numbers 12 and 16, you would type

```
12 + 16
```

You can't just put a mathematical operation like this on a line by itself, however. You can do two things:

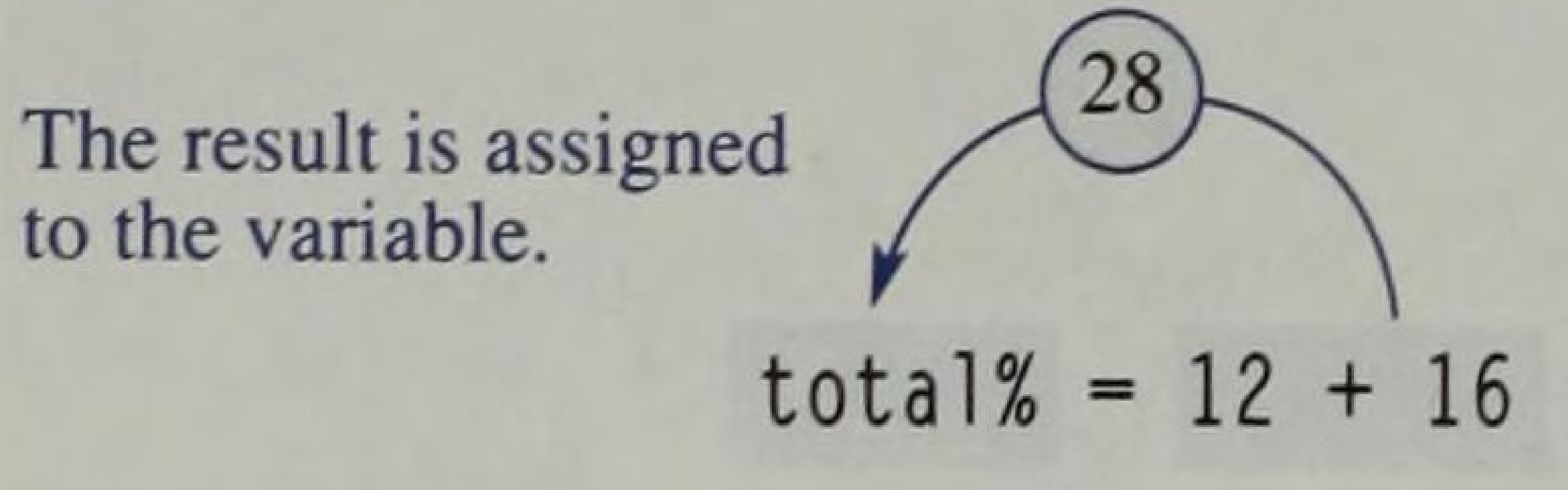
- Assign the result of the mathematical operation to a numeric variable:

  ```
  total% = 12 + 16   'Assign the result to a variable
  ```

- Use the result of the mathematical operation as the argument for a QBasic statement:

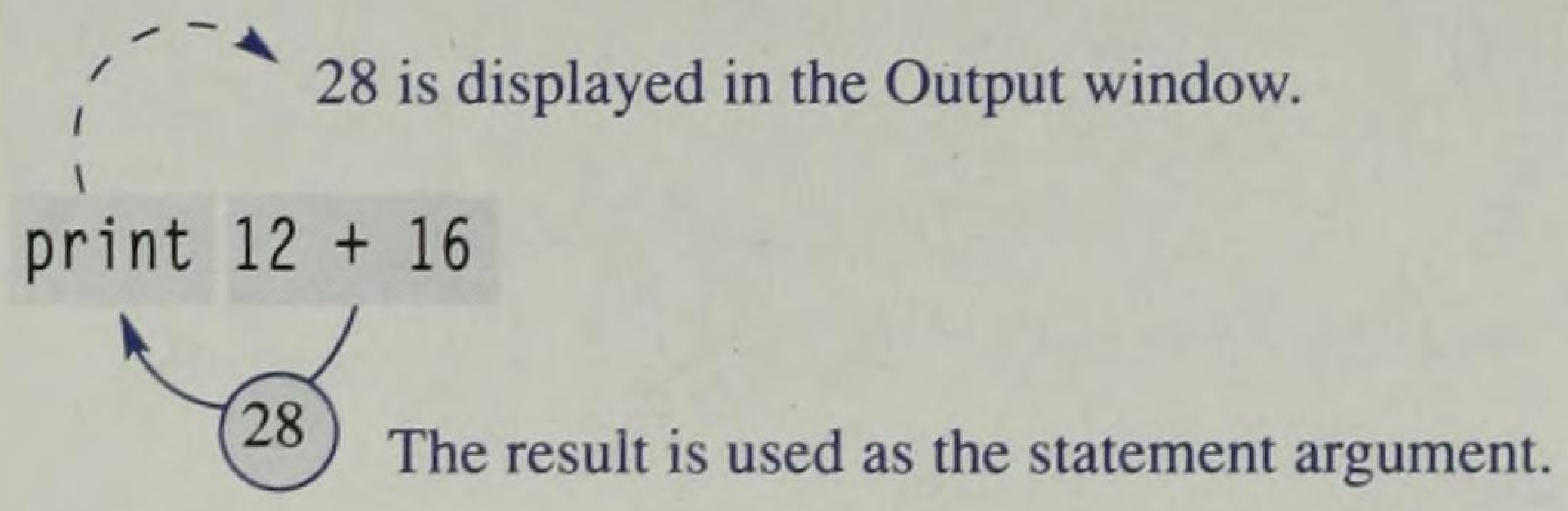
  ```
  PRINT 12 + 16      'Print the result directly
  ```

When you assign a mathematical operation to a variable, you're assigning the *result* of the operation to the variable, as shown here:


The result is assigned to the variable.

```
total% = 12 + 16
```

When you use a mathematical operation as a statement argument, you're using the *result* of the operation as the statement argument, as shown here:


28 is displayed in the Output window.

```
print 12 + 16
```

The result is used as the statement argument.

Practice: Working with QBasic operators

In Chapter 2, you learned that the Immediate window is a convenient place to try out a program line before you put it in your program. We'll use the Immediate window to experiment with QBasic operators.

Using the Immediate Window

The Immediate window lets you test individual program lines before you type them into the View window. The following hints will help you use the Immediate window effectively:

- You can execute only one line at a time in the Immediate window.
- To run a line in the Immediate window, simply press Enter. You needn't choose Run.
- You can't save any lines you enter in the Immediate window unless you use the editing commands to move the lines to the View window.

1. Press F6 to make the Immediate window the active window, and then enter the following line:

   ```
   CLS
   ```

2. Press any key to return to the Immediate window, and then enter the following line:

   ```
   PRINT 7 * 7
   ```

   Your output screen will display the result of the PRINT statement:

   ```
    49
   ```

3. Press any key to return to the Immediate window, and then enter the following line:

   ```
   PRINT 75 / 6
   ```

   The output screen will display the result of the PRINT statement:

   ```
    12.5
   ```

The \, MOD, and ^ Operators

QBasic has three special-purpose operators: \, MOD, and ^. These operators provide useful complements to the more familiar addition, subtraction, multiplication, and division operators.

The \ (integer division) operator

The integer division operator (\) works exactly as the standard division operator (/), except that when it divides one number into another it discards any fractional part of the result and returns only the integer part.

Practice: Using the \ operator

Enter the following line in the Immediate window:

```
PRINT 5 \ 2
```

Your output screen will display the result of the operation:

```
 2
```

Whereas standard division would provide a result of 2.5 or 2 with a remainder of 1, integer division discards the remainder and gives you the result 2.

The MOD operator

Although MOD doesn't look like a typical single-symbol operator, it is in fact a legitimate QBasic operator.

The MOD operator is also called the *remainder* operator. Its result is the opposite of the integer division (\) operator's: When you divide one number into another with the MOD operator, QBasic discards the integer portion of the result and returns only the remainder.

Practice: Using the MOD operator

Enter the following line in the Immediate window:

```
PRINT 5 MOD 2
```

Your output screen will display the result of the operation:

```
 1
```

Again, with standard division the result of this operation would be 2 with a remainder of 1. Because you used the MOD operator, QBasic discarded the integer portion of the result and returned only the remainder.

Different Rules for Division

The standard division operator (/) fully divides one number by another. For example, the following statement displays a result of 2.5—the same result a calculator provides:

```
PRINT 5 / 2
```

The \ and MOD operators, however, use the more traditional, or "long," method of division. That is, the results given by these two operators take into account an evenly divided part and a remainder part.

The ^ (exponentiation) operator

The exponentiation operator (^) lets you raise a number to a power of itself. For example, the QBasic equivalent of 10^3 (ten to the third power) looks like this:

```
10 ^ 3
```

Practice: Using the ^ operator

Enter the following line in the Immediate window:

```
PRINT 10 ^ 3
```

Your output window will display the result of the operation:

```
 1000
```

Numeric Expressions

A *numeric expression,* commonly called a formula, is simply a combination of numbers, numeric variables, numeric operators, and numeric functions that collectively yield a result. The multiplication, division, and exponentiation exercises you just did were all simple examples of numeric expressions.

QBasic allows you to create a wide range of numeric expressions, from the simple ones you just practiced to elaborate ones. Best of all, creating numeric expressions isn't complicated. All you need to do is to learn a few basic rules, and you'll be able to calculate almost anything.

Using more than one operator

The examples you just worked with all used a single operator. QBasic will let you use more than one operator in the same numeric expression, allowing you to do several calculations at once.

Practice: Using more than one operator

Enter the following statement in the Immediate window:

```
PRINT 14 + 26 + 15.75 - 33.2
```

The output screen should display the result of the operation:

```
 22.55
```

As you can see, the Immediate window can come in very handy as a calculator!

Order of calculation

QBasic follows a strict set of rules when it calculates a numeric expression that contains more than one operator. Consider the following numeric expression:

$$3 + 4 * 5$$

What's the result of this expression? Actually, there are two different results, depending on how you calculate it. If you perform addition first, the answer is 35. If you perform multiplication first, the answer is 23.

To avoid such confusion, QBasic calculates operations in the following order:

- Exponentiation (^) is performed first.
- Multiplication and division (*, /, \, and MOD) are performed next.
- Addition and subtraction (+ and –) are performed last.

These rules don't cover all circumstances, however. What about expressions in which operators are all of the same type or "level of importance"? For example,

*3 * 5 MOD 2*

or

*100 / 4 * 3*

When QBasic encounters such expressions, it calculates them from left to right.

Practice: Working with operator precedence

Enter the following statement in the Immediate window:

```
PRINT 3 + 4 * 5
```

Your output screen will display the result of the operation:

```
 23
```

Because QBasic does the multiplication before the addition, the result is 23. Press any key to return to the Immediate window, and then enter the following statement:

```
PRINT 100 / 4 * 3
```

Your output screen will display the result of the operation:

```
 75
```

Because the division (/) and multiplication (*) operators have the same weight—that is, the same priority in the list of operator precedence—QBasic calculates the expression from left to right.

Using parentheses to control the order of calculation

Consider the numeric expression you looked at earlier:

*3 + 4 * 5*

As you just learned, QBasic performs the multiplication before it performs the addition, resulting in the value 23. But what if you *want* the addition to be performed before the multiplication?

QBasic lets you control how a numeric expression is calculated. If you put part of a numeric expression inside *parentheses,* QBasic performs that part of the calculation before any other. For example, the following numeric expression produces a result of 35 because QBasic calculates the contents of the parentheses before it calculates the rest of the expression:

_(3 + 4) * 5_

If you use two or more operators within the same set of parentheses, QBasic calculates the contents of the parentheses using the standard rules. For example, the following numeric expression produces a result of 23:

_(3 + 4 * 5)_

Practice: Using parentheses to control how a numeric expression is calculated

1. Enter the following statement in the Immediate window:

   ```
   PRINT 3 + 4 * 5
   ```

   Your output screen will display the result of the operation:

   ```
    23
   ```

2. Press any key to return to the Immediate window, and then enter the following statement:

   ```
   PRINT (3 + 4) * 5
   ```

   Your output screen will display the result of the operation:

   ```
    35
   ```

Using parentheses within parentheses

If you need to use a numeric expression that requires more control than separate sets of parentheses can provide, you can use *nested parentheses*—that is, one set of parentheses within another.

Consider the following numeric expression:

_2 + 3 * 4 ^2_

QBasic would calculate the exponentiation first, then the multiplication, and finally the addition. But what if you wanted QBasic to perform the addition first, then the multiplication, and then the exponentiation? If you used

`(2 + 3 * 4) ^ 2`

QBasic would calculate the contents of the parentheses first and do the exponentiation last, but it would still perform the multiplication before the addition. The answer would be to use a set of nested parentheses, like this:

`((2 + 3) * 4) ^ 2`

When QBasic encounters nested parentheses, it calculates the contents of the innermost parentheses before calculating the contents of the outermost parentheses. In the example above, QBasic would perform the addition first because it's in the innermost parentheses, then the multiplication, and finally the exponentiation.

QBasic's Mathematical Functions

As you learned in Chapter 3, a function returns a value to a program. QBasic provides you with several mathematical functions. (As with any other QBasic function, you must use these functions within a QBasic statement.)

The symbol *n* in the table on the next page represents the number, numeric variable, or expression upon which you want the function to operate. Notice that *n* is enclosed in parentheses. You must use these parentheses when you provide a value as an argument to a function.

Practice: Working with QBasic's numeric functions

Enter the following statement in the Immediate window:

```
PRINT SQR(36)
```

Your output screen should display the result of the operation, in this case the square root of 36:

```
 6
```

Function	***Purpose***
ABS(*n*)	Returns the absolute value of *n*
ATN(*n*)	Returns the arctangent, in radians, of *n*
COS(*n*)	Returns the cosine of angle *n* expressed in radians
EXP(*n*)	Returns the logarithm of *n*
SGN(*n*)	Returns −1 if *n* is less than zero, returns 0 if *n* is zero, and returns +1 if *n* is greater than zero
SIN(*n*)	Returns the sine of angle *n* expressed in radians
SQR(*n*)	Returns the square root of *n*
TAN(*n*)	Returns the tangent of angle *n* expressed in radians

Using mathematical functions in numeric expressions

You can use mathematical functions in a numeric expression along with other QBasic operators. In the following example, the SQR function solves a problem involving a right triangle.

Practice: Using the SQR function

Suppose you have a pole 12½ feet high and a metal stake in the ground 15 feet from the base of the pole (Figure 4-4). You plan to string a cable from the top of the pole to the stake, but you aren't sure how much cable to buy.

You can write a QBasic program that uses the SQR function to calculate the length for you. The formula for this calculation is

$$\sqrt{height^2 + length^2}$$

1. Press the F6 function key to make the View window the active window.
2. Select the New command from File menu, and then type in the following program.

```
CLS

PRINT "This program calculates how much cable you need to"
PRINT "run from the top of a pole to a stake in the ground."
PRINT
INPUT "Enter the height, in feet, of the pole:  ", height!
PRINT
PRINT "Enter the length, in feet, from the base"
INPUT "of the pole to the stake in the ground:  ", length!

cable! = SQR((height! ^ 2) + (length! ^ 2))

PRINT
PRINT "You need to buy"; cable!; "feet of cable."
```

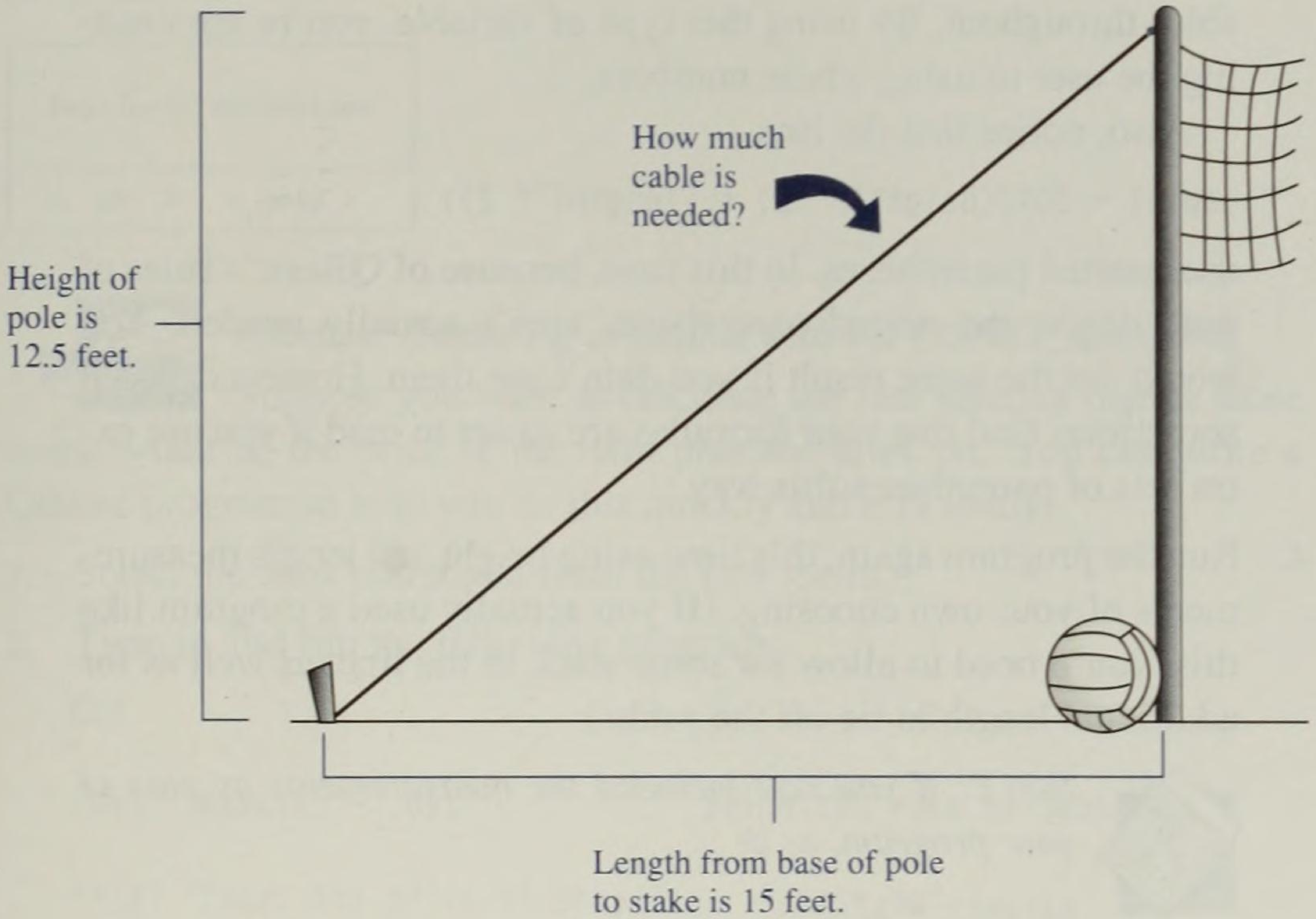

How much cable is needed?
Height of pole is 12.5 feet.
Length from base of pole to stake is 15 feet.

FIGURE 4-4.
Finding the length of a cable.

3. Run the program, and enter the height and length measurements given in Figure 4-4—12.5 and 15, respectively. Your output will look like this:

   ```
   This program calculates how much cable you need to
   run from the top of a pole to a stake in the ground.

   Enter the height, in feet, of the pole:  12.5

   Enter the length, in feet, from the base
   of the pole to the stake in the ground:  15

   You need to buy 19.52562 feet of cable.
   ```

   Notice that the program uses single-precision floating-point variables throughout. By using this type of variable, you're not limiting the user to using whole numbers.

   Also, notice that the line

   ```
   cable! = SQR((height! ^ 2) + (length! ^ 2))
   ```

   uses nested parentheses. In this case, because of QBasic's rules of precedence, the nested parentheses aren't actually needed. You would get the same result if you didn't use them. However, you'll sometimes find that your formulas are easier to read if you use extra sets of parentheses this way.

4. Run the program again, this time using height and length measurements of your own choosing. (If you actually used a program like this, you'd need to allow for some slack in the line, as well as for additional length to tie off the cable.)

   NOTE: If you had included the measurements as part of your program, as in

   ```
   height! = 12.5
   length! = 15
   ```

   you would have had to change the program itself if you wanted to use another set of measurements. By allowing the measurements to be entered by the user, however, you can run the program again and again without having to change anything.

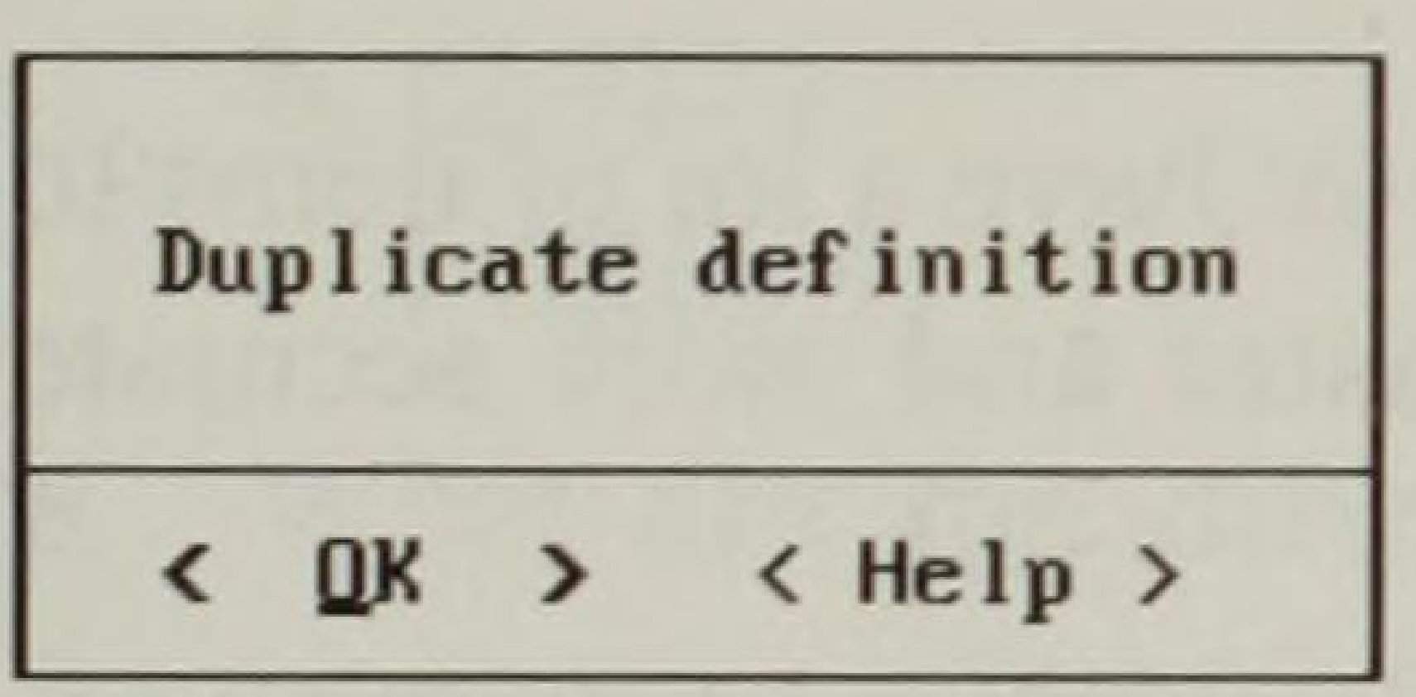
THE CONST STATEMENT

When you have a value that will never change throughout a program, such as a local or state sales tax or a discount rate, declare it as a constant with the CONST statement.

To use the CONST statement, simply place the keyword before the constant declaration. For example, the following statement tells QBasic that you want the single-precision floating-point constant named *TAXRATE!* to always represent a value of 0.081:

```
CONST TAXRATE! = .081
```

If you try to assign another value to *TAXRATE!* later in the program, QBasic will display the following dialog box when you try to run the program:



Duplicate definition

Practice: Declaring constants with the CONST statement

Suppose you want to calculate the *real* price of one or more items—that is, the price of the item plus the sales tax. You can write a QBasic program to help you do this quickly and efficiently.

1. Select the New command from the File menu.
2. Type in and run the following program:

```
CLS

CONST TAXRATE! = .081           ' Sales tax rate in Seattle

PRINT "Enter the price of the item.  Please do"
INPUT "not use a dollar sign or any commas:  $", price!

tax! = price! * TAXRATE!        ' Calculate the tax amount
total! = price! + tax!          ' Calculate the total cost
```

(continued)

continued

```
PRINT
PRINT "Cost of the item is $"; price!
PRINT "Sales tax would be $"; tax!
PRINT "----------------------------"
PRINT "Total cost would be $"; total!
```

Your output screen should look something like this:

```
Enter the price of the item.  Please do
not use a dollar sign or any commas:  $2500

Cost of the item is $ 2500
Sales tax would be $ 202.5
----------------------------
Total cost would be $ 2702.5
```

You can run this program as often as you like. If you have a lot of items to calculate, you'll find this program to be much faster and more accurate than a calculator.

This program also demonstrates that you can use variables and constants to create new variables. Notice the lines

```
tax! = price! * TAXRATE!          ' Calculate the tax amount
total! = price! + tax!            ' Calculate the total cost
```

There are no numbers involved here, only numeric variables and a constant. Because QBasic treats numeric variables and constants just as it would numbers, you can use them anywhere you would normally use a number.

SUMMARY

Congratulations! You have just taken a major step forward in your journey toward becoming a QBasic programmer. Variables (both string and numeric) and operators are important topics—topics that find their way into the very heart of most QBasic programs. In the next chapter, you'll add some intelligence to your programs by allowing them to make decisions on their own, based on sets of rules you give them.

QUESTIONS AND EXERCISES

1. What are the four types of QBasic numeric variables, and how do they differ?
2. Why does QBasic put a space in front of a positive number?
3. What does it mean when QBasic displays an *Expected: end-of-statement* dialog box?
4. What does it mean when QBasic displays an *Overflow* dialog box or error message?
5. What type of variable would you use for each of the following numbers?

   a. –32679
   b. 12.3774
   c. 142286.9
   d. 14.000001
   e. –1286.0
   f. –268.0005
   g. 268110
   h. –10.222222
   i. –.0000001

6. What are the differences between regular division with the / operator, integer division with the \ operator, and remainder division with the MOD operator?
7. From highest priority to lowest priority, what is the order of precedence that QBasic applies to mathematical operators?
8. What is the result of this calculation?

```
(((5 + 8) - (1 + 3) / 4) * ((7 - 2) ^ 2))
```

9. Write a program (complete with comments) that calculates and prints the following values:

   ☐ ABS(–10) + 5

   ☐ SQR(36)

   ☐ SQR(4) ^ 2

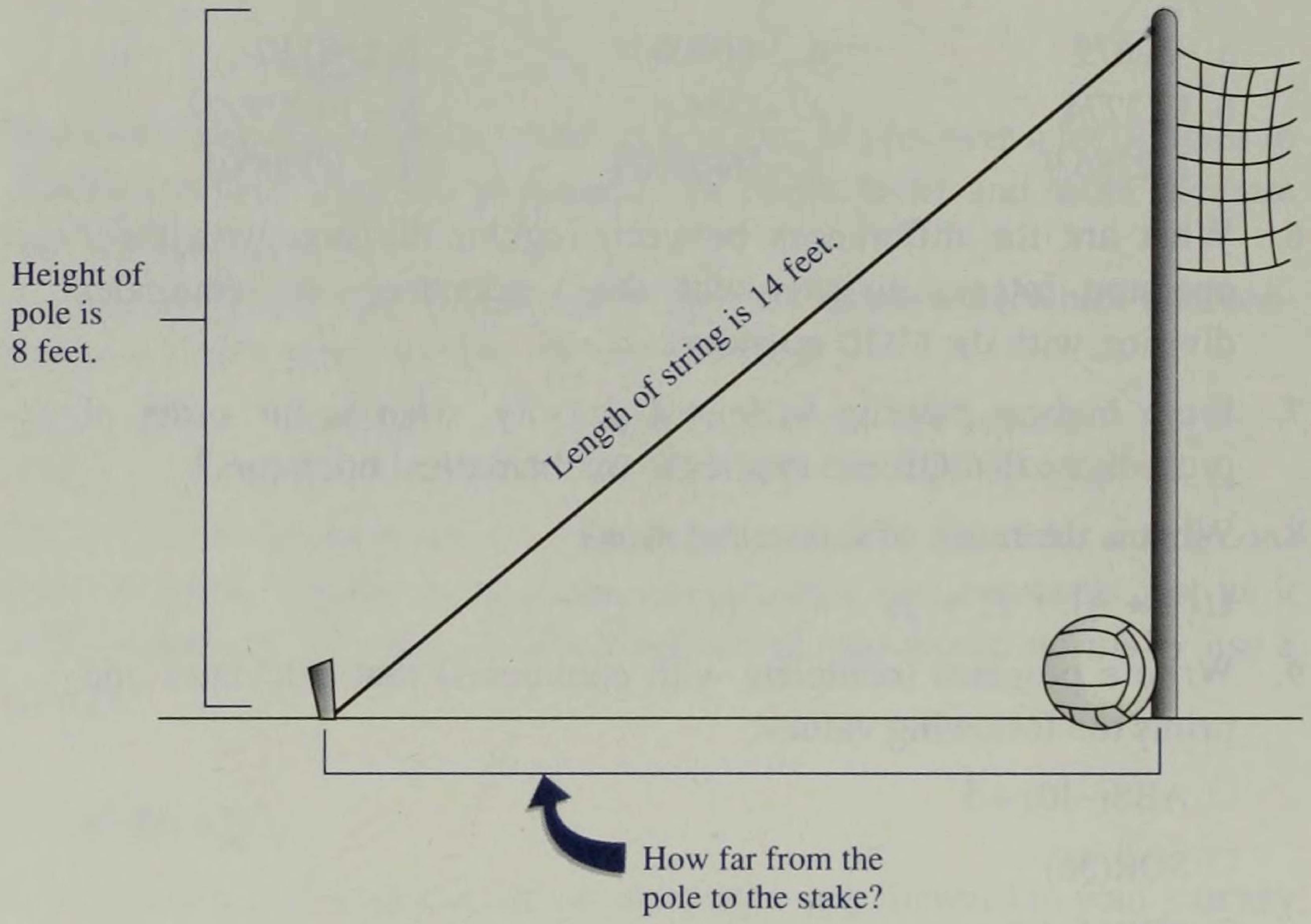
   ☐ COS(3.141592654)

10. The value of the mathematical constant π can be approximated as 3.141592654. The formula for the circumference of a circle is

    $2 \times \pi \times radius$

    (*radius* is the distance from the center of the circle to the edge of the circle.) Write a program that asks the user for the radius of a circle and then displays a message telling the user the circumference of the circle.

11. (BONUS) You're setting up a volleyball net. The top of the pole is 8 feet off the ground, and the string tied to the top of the pole is 14 feet long:


Height of pole is 8 feet.
Length of string is 14 feet.
How far from the pole to the stake?

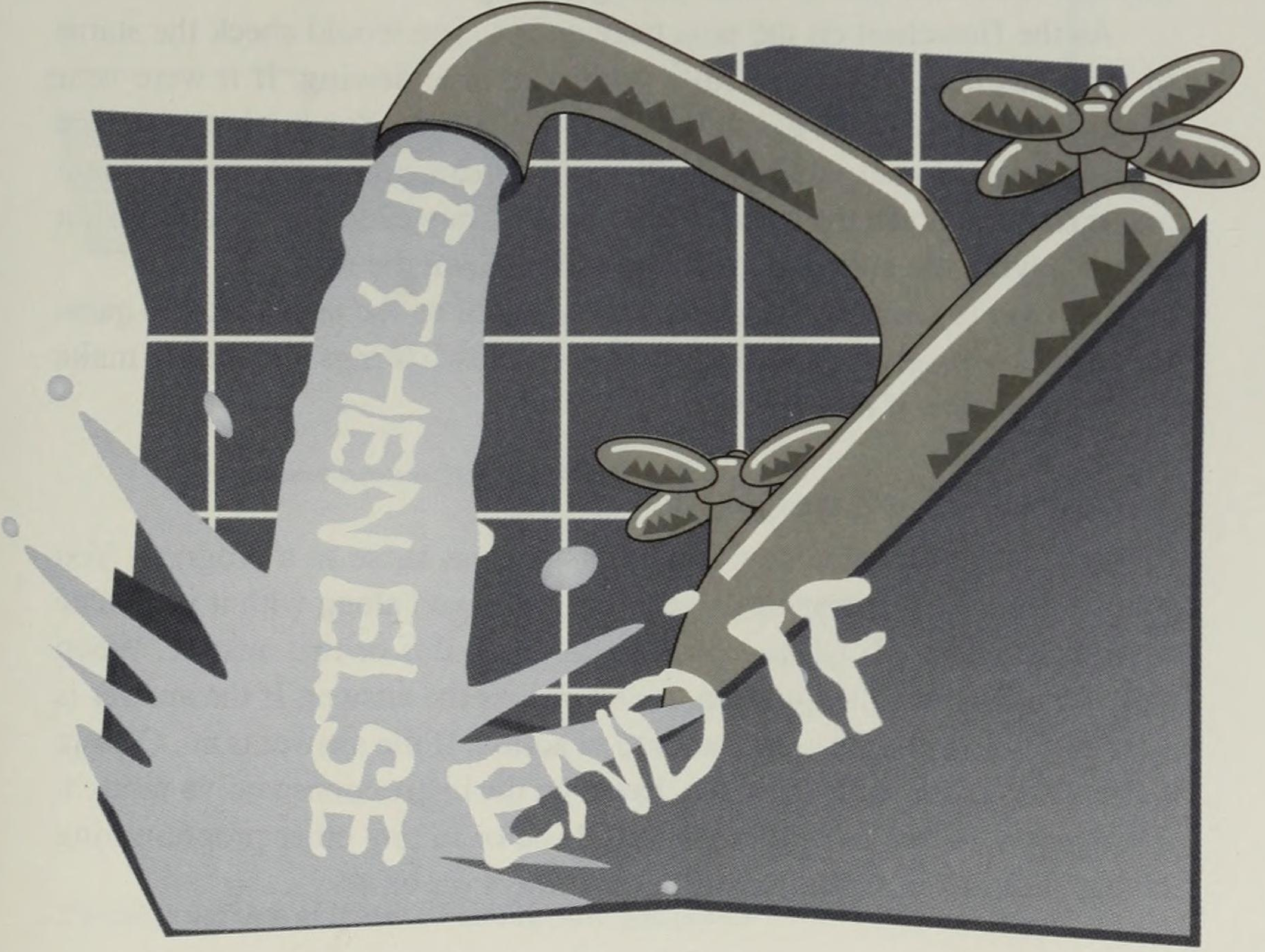
The formula for calculating the distance from the bottom of the pole to where the stake should go is

$\sqrt{string\_length^2 - height^2}$

Write a QBasic program that asks the user for the height of the pole and the length of the string and then prints out how many feet away from the bottom of the pole the stake should be driven.

CHAPTER 5

Controlling Program Flow


IF THEN ELSE
END IF

The programs you've written thus far have all run in a very straightforward way: QBasic runs the first instruction and then works progressively toward the final instruction. At times, however, you might want QBasic to run certain parts of your program under a certain set of circumstances. This chapter teaches you how to program in this way.

INTRODUCTION TO DECISION MAKING

You make thousands of decisions every day. Some decisions require that you weigh your options; others require little or no thought. The decision whether to breakfast on a fruit cup or a jelly doughnut might give you pause, but the decision to reach up and scratch your ear is likely to be spontaneous.

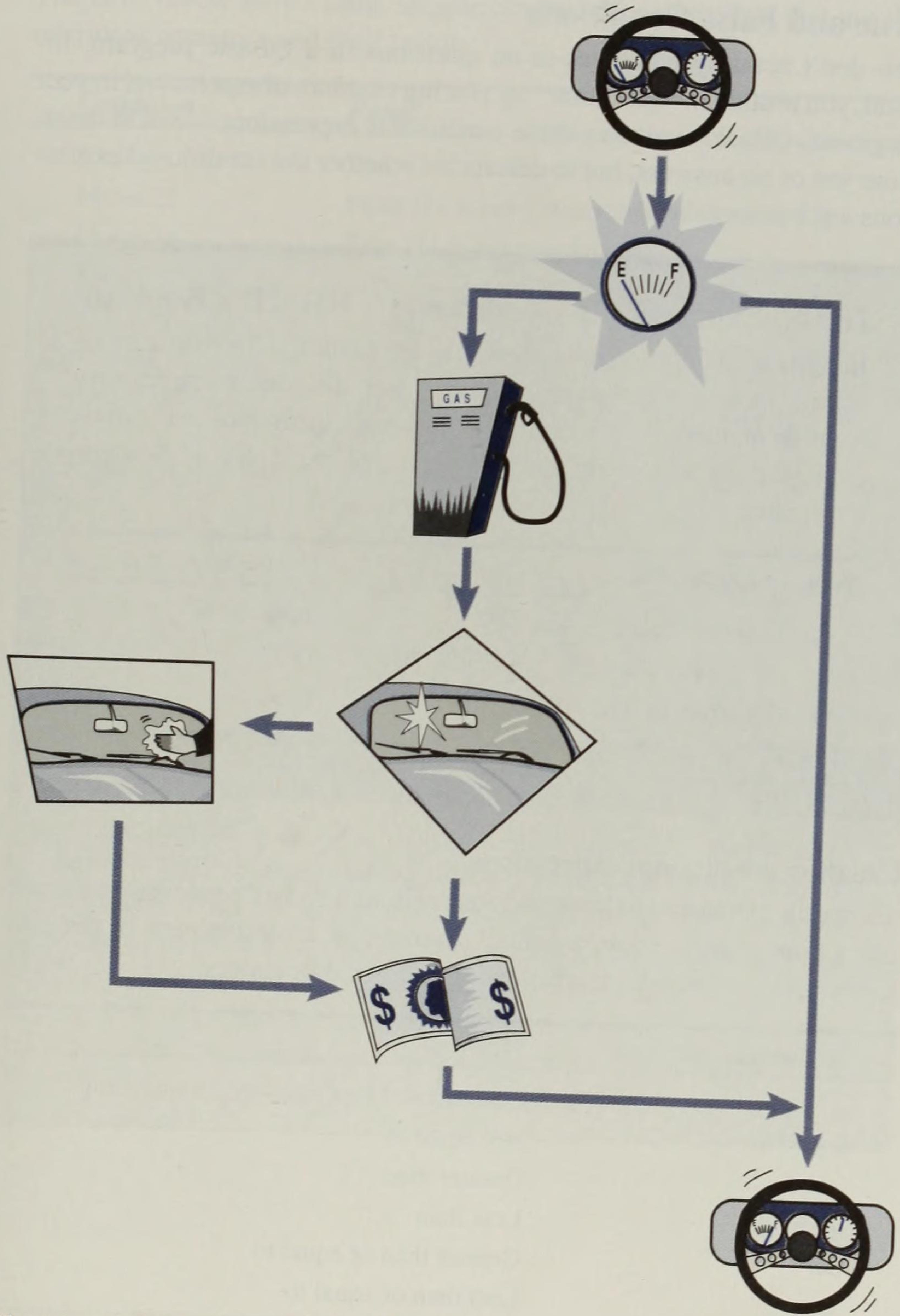
Other decisions can lead to separate sets of actions. For example, if you're on a long driving trip, you need to be sure you don't run out of gas. Let's examine a typical decision-making process, one you would go through each time you checked your gas gauge.

As the flowchart on the next page shows, you would check the status of your gas tank. If it were full, you'd continue driving. If it were near empty, you'd pull into a gas station and fill the tank. You might then check the condition of your windshield. If it were clean, you'd pay the attendant and then drive down the road. If it were dirty, however, you would wash it before paying the attendant and continuing down the road.

It's easy to see how a single decision, based on the answer to the question *Am I low on gas?*, caused you to do one of two things and then to make further decisions if necessary.

Decision Making in QBasic

QBasic lets you include decision points such as these in a program. You put a questionlike statement in your program, and along with it you write instructions that tell QBasic what to do given this or that answer. When QBasic encounters the question, it determines the answer. If the answer is yes, QBasic takes a particular course of action. If the answer is no, QBasic takes a different course of action, based on the instructions you've written. The answer *maybe* never occurs in QBasic or in any other programming language. The answer to a question is always *yes* or *no.*



Decision making at work: Checking your gas gauge.

True and False Conditions

You don't actually place yes-or-no questions in a QBasic program. Instead, you establish ''questions'' by placing *conditional expressions* in your programs. QBasic evaluates these conditional expressions—not to determine yes or no answers, but to determine whether the conditional expressions are *true* or *false*.

IF the Expression's Conditional THEN It's Boolean

In QBasic, conditional expressions are actually *Boolean expressions*. Boolean expressions, named after the nineteenth-century English mathematician George Boole, are expressions that can be evaluated as *true* or *false*. Here are some examples of Boolean expressions:

Boolean expression	*Evaluation*
A pint is larger than a gallon.	False
Twelve is greater than ten.	True
Five is less than or equal to six.	True
Eleven inches are equal to one foot.	False

Creating conditional expressions

To create conditional (Boolean) expressions in your program, you must use a *relational operator*, a *logical operator*, or a combination of the two kinds. QBasic provides the following relational operators:

Relational operator	*Meaning*
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The table below shows some sample conditional expressions that use the relational operators and their results:

Condition	*Result*
3 < 7	True (3 is less than 7)
14 >= 22	False (14 is not greater than or equal to 22)
11 <> 16	True (11 is not equal to 16)
5 = –5	False (positive 5 is not equal to negative 5)
12 > 14	False (12 is not greater than 14)
11 >= 11	True (11 is greater than or equal to 11)
total% < 5	True if the value of *total%* is less than 5; otherwise, false
num1% = num2%	True if the value of *num1%* is equal to the value of *num2%*; otherwise, false

You'll get a chance to work with relational operators and conditional expressions shortly. Right now, let's take a look at some of the QBasic statements that allow you to use these conditional expressions in your programs.

Numeric and Conditional Expressions

In the last chapter, you learned about *numeric* expressions. Numeric expressions and conditional expressions look similar: Both consist of an operator and data. The difference between a numeric expression and a conditional expression is that a numeric expression uses a numeric operator (such as +, –, /, or MOD) and a conditional expression uses a conditional operator (such as >=, <, <>, or <=). A numeric expression yields a numeric result, but a conditional expression yields a true-or-false result.

THE IF STATEMENT

The IF statement lets you evaluate a condition and works together with the THEN clause to take a course of action based on the evaluation.

In their simplest form, IF and THEN form a single statement. The syntax of a single-line conditional statement using IF and THEN is as follows:

```
IF condition THEN statement
```

The *condition* portion of the statement is one of the conditional expressions you just learned about. The *statement* portion is another QBasic statement that is executed only if *condition* is true. If *condition* is false, QBasic ignores *statement* and moves on to the next line in your program. Here's an example:

If this condition is true, QBasic executes this statement.

```
IF userNum% > 1000 THEN PRINT "The number is too big!"
```

If this condition is false, QBasic ignores the rest of the line and executes the next statement in your program.

The statement following THEN can be any legal QBasic statement. But remember that QBasic executes the statement following THEN *only* if the condition following IF is true.

Saving Your Practice Programs

From this point on, programs in *Running MS-DOS QBasic* appear in colored boxes and are named. We recommend that you type these programs in as you go in order to practice the QBasic programming concepts as we discuss them. When you've finished, save the programs on disk for future reference. Building a collection of your own programs and revising routines from them will be a real time-saver for you as you tackle larger programming projects.

Practice: Using the IF statement

1. Type in the IF-1.BAS program (Figure 5-1) and run it.

```
' IF-1.BAS
' This program demonstrates the IF statement.

CLS

INPUT "How many Rockettes dance at Radio City Music Hall?  ", guess%
IF guess% = 36 THEN PRINT "That's right!!"
IF guess% <> 36 THEN PRINT "Sorry!  The correct answer is 36!"
PRINT "Radio City Music Hall opened on December 27, 1932."
```

FIGURE 5-1.
IF-1.BAS: A program demonstrating the IF statement.

2. Enter the number *36*. Your output screen will look like this:

   ```
   How many Rockettes dance at Radio City Music Hall?  36
   That's right!!
   Radio City Music Hall opened on December 27, 1932.
   ```

   Because you entered the value *36*, the condition in the first IF statement of the program was true, so QBasic executed the PRINT statement at the end of the line. Also, because the condition in the second IF statement was *not* true, QBasic did not execute the PRINT statement at the end of the second IF statement.

3. Run the program again, entering a value other than 36. Your output screen should look something like this:

   ```
   How many Rockettes dance at Radio City Music Hall?  4
   Sorry!  The correct answer is 36!
   Radio City Music Hall opened on December 27, 1932.
   ```

   This time, the value you entered caused a different set of actions to occur. Because the value *4* is not equal to the value *36*, the condition in the first IF statement was false, so QBasic ignored the rest of that statement. However, this time the condition in the second IF statement was true, so QBasic did execute the remainder of that statement.

Notice that in both cases the final PRINT statement of the program printed the message *Radio City Music Hall opened on December 27, 1932.* This demonstrates that even if the condition in an IF statement is false, QBasic executes the rest of the program as it normally would. Only the statement that follows the THEN portion of an IF statement is ignored if the condition is not true.

Using More Than One Condition with IF

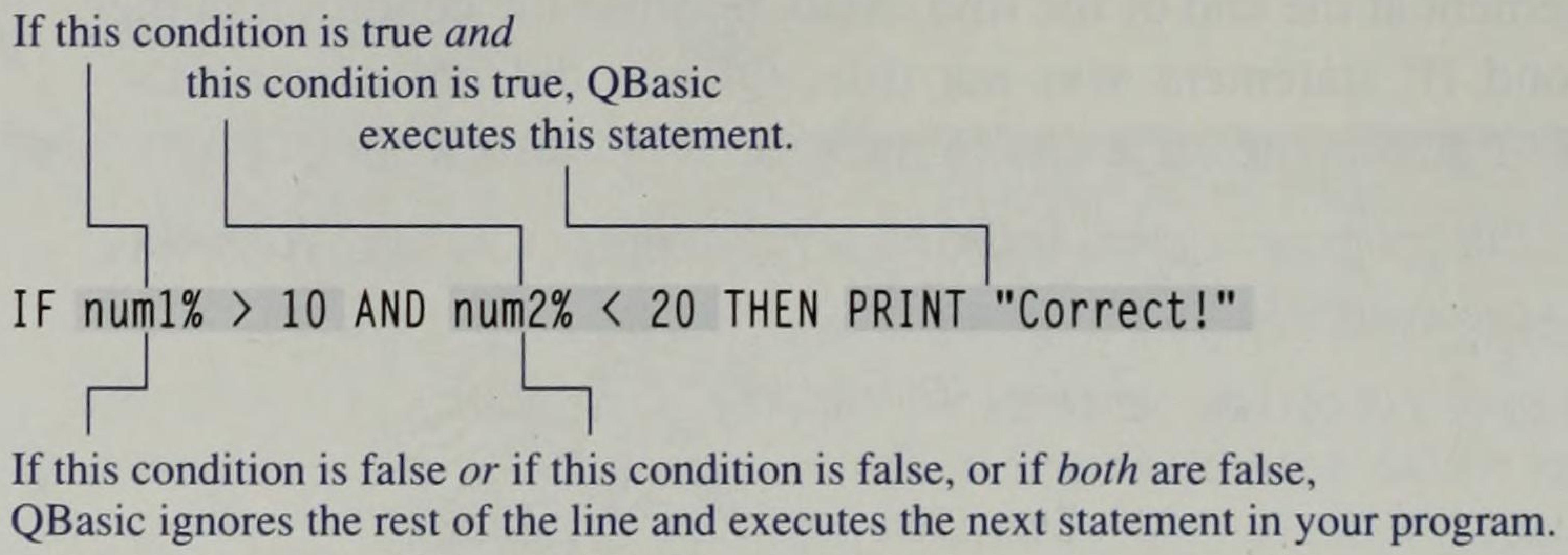
In the preceding program, QBasic evaluated a single condition in each IF statement. You can specify multiple conditions in an IF statement by using the *logical operators* AND and OR. You use logical operators in conditional expressions much as you use math operators in numeric expressions.

The AND logical operator

The AND operator lets you specify multiple conditions that must be true before an action can be taken. Here's the syntax line for an IF statement that uses the AND operator:

```
IF condition1 AND condition2 THEN statement
```

Both *condition1* and *condition2* must be true before QBasic can execute *statement.* Here's an example:



If this condition is true *and*
this condition is true, QBasic
executes this statement.

```
IF num1% > 10 AND num2% < 20 THEN PRINT "Correct!"
```

If this condition is false *or* if this condition is false, or if *both* are false,
QBasic ignores the rest of the line and executes the next statement in your program.

Note that the IF statement contains two conditions and that because they're connected by the logical operator AND, both *num1% > 10* and *num2% < 20* must be true before QBasic can execute the PRINT statement following THEN.

Practice: Working with the AND operator

1. Type in the IF-2.BAS program (Figure 5-2) and run it.

```
' IF-2.BAS
' This program demonstrates the AND logical operator.

CLS

INPUT "How many teenagers can fit in a phone booth?  ", guess%
PRINT
IF guess% > 9 AND guess% < 13 THEN PRINT "That's right!!"
PRINT "Depending on their size, approximately 10 to 12"
PRINT "teenagers can fit in a phone booth."
```

FIGURE 5-2.
IF-2.BAS: A program demonstrating the AND logical operator.

2. Enter the number *10*. Your output screen will look like this:

```
How many teenagers can fit in a phone booth?  10

That's right!!
Depending on their size, approximately 10 to 12
teenagers can fit in a phone booth.
```

   Because you entered the value *10*, both conditions in the IF statement were true. If you run the program again and enter an integer value less than 10 or more than 12, one of the conditions will be false and QBasic will not execute the PRINT statement at the end of the IF statement.

The OR logical operator

The OR operator lets you create a more flexible set of conditions that must be met before an action can take place. The syntax line for an IF statement using the OR operator is as follows:

```
IF condition1 OR condition2 THEN statement
```

Note that the IF statement contains two conditions. Only *one* of these conditions need be true before QBasic can execute the statement following

THEN. QBasic also executes the statement following THEN if both conditions are true. Here's an example:

If this condition is true *or*
this condition is true, QBasic
executes this statement.

```
IF quota% > 10 OR sales% > 1000 THEN PRINT "Good job!"
```

If both of these conditions are false, QBasic ignores the rest of the line
and executes the next statement in your program.

Practice: Working with the OR operator

1. Type in the IF-3.BAS program (Figure 5-3).

```
' IF-3.BAS
' This program demonstrates the OR logical operator.

CLS

PRINT "I'm thinking of a number between 1 and 100."
PRINT "Can you guess what it is?"
INPUT "Please enter a number between 1 and 100:  ", guess%
PRINT
IF guess% = 63 THEN PRINT "That's right!!"
IF guess% < 53 OR guess% > 73 THEN PRINT "You're way off!"
PRINT "Thanks for playing!"
```

FIGURE 5-3.
IF-3.BAS: A program demonstrating the OR logical operator.

2. Run the program. Enter the value *63*, the number the program is "thinking of." Your output screen will look like this:

```
I'm thinking of a number between 1 and 100.
Can you guess what it is?
Please enter a number between 1 and 100:  63

That's right!!
Thanks for playing!
```

3. Run the program again, and use a number that is less than 53 or greater than 73. Your output screen will look something like this:

```
I'm thinking of a number between 1 and 100.
Can you guess what it is?
Please enter a number between 1 and 100:  76

You're way off!
Thanks for playing!
```

Look at the second IF statement. By entering a value that was 11 or more less than or greater than the number the program was "thinking of," you caused QBasic to print the message *You're way off!*

The NOT logical operator

The NOT operator lets you *negate* a condition: If a condition is false, the NOT operator makes the condition true; if a condition is true, NOT makes it false. Here's the syntax for an IF statement using the NOT operator:

IF NOT *condition* THEN *statement*

NOT is useful when you want to execute a statement when a condition is *not* true. Here's an example:

If this condition is false, QBasic executes this statement.

```
IF NOT age% >= 18 THEN PRINT "You can't vote."
```

If this condition is true, QBasic ignores the rest of the line and executes the next statement in your program.

Using ELSE with IF and THEN

Now you know how to make QBasic evaluate a condition and take an action if the condition is true or if the condition is false. But what if you want QBasic to choose between *two* actions based on the condition?

When paired with IF and THEN, ELSE lets you specify two separate actions for QBasic: one action (following THEN) if the condition is true, and a different action (following ELSE) if the condition is false.

Here's the syntax for an IF statement using THEN and ELSE:

IF *condition* THEN *statement1* ELSE *statement2* where *condition* is the logical condition you want QBasic to evaluate, *statement1* is the BASIC statement QBasic executes if *condition* is true, and *statement2* is the BASIC statement QBasic executes if *condition* is false. IF, THEN, and ELSE must all appear on the same line. Here's an example:

If this condition is true, QBasic executes this statement.

```
IF userNum% > 1000 THEN PRINT "True" ELSE PRINT "False"
```

If this condition is false, QBasic executes this statement.

Practice: Working with IF, THEN, and ELSE

1. Type in the IF-4.BAS program (Figure 5-4).

```
' IF-4.BAS
' This program demonstrates the ELSE statement.

CLS

PRINT "I’m thinking of a number between 1 and 5."
PRINT "Can you guess what it is?"
INPUT "Enter a number between 1 and 5:  ", guess%
PRINT
IF guess% = 3 THEN PRINT "That's right!!" ELSE PRINT "Sorry!"
PRINT "Thanks for playing!"
```

FIGURE 5-4.
IF-4.BAS: A program demonstrating the ELSE clause.

2. Run the program and enter the number *3*. Your output screen will look like this:

```
I’m thinking of a number between 1 and 5.
Can you guess what it is?
Enter a number between 1 and 5:  3

That's right!!
Thanks for playing!
```

Because you entered the value *3*, the condition is true, so QBasic executed the PRINT statement following THEN.

3. Run the program again, entering a value other than 3. Your output screen will look something like this:

```
I'm thinking of a number between 1 and 5.
Can you guess what it is?
Enter a number between 1 and 5:  4

Sorry!
Thanks for playing!
```

This time the condition was not true, so QBasic executed the PRINT statement following ELSE.

Making Longer Conditional Statements with END IF

Using ELSE with IF and THEN is a handy way to give QBasic two options to take depending on the evaluation of a particular condition. However, as you just saw, packing all that information on a single line doesn't allow much room for creativity.

That's where END IF comes in—you can create longer programs by placing each possible course of action on a separate line. Here's the syntax for an IF statement that uses THEN, ELSE, and END:

```
IF condition THEN
    statements executed if condition is true
ELSE
    statements executed if condition is false
END IF
```

This type of IF statement consists of several individual lines, collectively known as a *block*. The *condition* part of the statement is the conditional expression you want QBasic to evaluate. The THEN keyword ends the line. THEN must always appear in the same line as IF.

If *condition* is true, QBasic

1. Executes the statements between THEN and ELSE
2. Bypasses the statements between ELSE and END IF
3. Continues executing the program

You can include any number of statements between THEN and ELSE.

If *condition* is false, QBasic

1. Bypasses the statements between THEN and ELSE
2. Executes the statements between ELSE and END IF
3. Continues executing the program

You can include any number of statements between ELSE and END IF.

Here's an example:

If this condition is true, QBasic executes these statements.

```
IF choice$ = "YES" THEN
    PRINT "Yes sir!  The 1958 Edsel is the car for you!"
    PRINT "Power this, power that, and just plain fun!"
ELSE
    PRINT "Perhaps you'd care to look at some of the other"
    PRINT "sleek, modern cars from the 1958 model year."
END IF
```

If this condition is false, QBasic executes these statements.

Notice how the indentation of related statements keeps the program easy to read. You should develop the habit of using indentation in your own programs.

Practice: Working with IF, THEN, ELSE, and END IF

1. Type in the IF-5.BAS program (Figure 5-5).

```
' IF-5.BAS
' This program demonstrates the block IF statement.

CLS

PRINT "Welcome to Automobile Trivia!"
PRINT
PRINT "In what year did Karl-Friedrich Benz test-drive"
INPUT "the first successful gasoline-driven automobile?  ", guess%
PRINT
```

FIGURE 5-5. *(continued)*
IF-5.BAS: A program demonstrating the block IF statement.

FIGURE 5-5. *continued*

```
IF guess% = 1885 THEN
    PRINT "That's right!  You're quite a car buff!"
ELSE
    PRINT "No, he first drove it at Mannheim, Germany,"
    PRINT "in 1885.  (It was patented on January 29, 1886.)"
END IF
PRINT "Thanks for guessing!"
```

2. Run the program. Enter the value *1885*, which is the value that causes the condition following IF to be true. Your output screen will look like this:

```
Welcome to Automobile Trivia!

In what year did Karl-Friedrich Benz test-drive
the first successful gasoline-driven automobile?  1885

That's right!  You're quite a car buff!
Thanks for guessing!
```

3. Run the program again, but this time enter a value other than 1885. Your output screen will look something like this:

```
Welcome to Automobile Trivia!

In what year did Karl-Friedrich Benz test-drive
the first successful gasoline-driven automobile?  1776

No, he first drove it at Mannheim, Germany,
in 1885.  (It was patented on January 29, 1886.)
Thanks for guessing!
```

As you can see, using END IF with IF, THEN, and ELSE allows you to use entire blocks of statements.

The ELSEIF Keyword

ELSEIF is similar to ELSE in that it provides an alternative course of action if *condition* is false. With ELSEIF, however, you supply another condition for QBasic to evaluate. Here's the syntax for an IF statement that uses ELSEIF:

```
IF condition1 THEN
     statements executed if condition1 is true
ELSEIF condition2 THEN
     statements executed if condition2 is true
ELSEIF condition3 THEN
     statements executed if condition3 is true
⋮
ELSE
     statements executed if all conditions are false
END IF
```

The column of dots between ELSEIF and ELSE in the preceding syntax indicates that you can have more ELSEIF statements followed by other conditions if you want to. There is no limit to the number of ELSEIF statements and associated conditions you can use.

NOTE: You must use THEN at the end of an ELSEIF statement. The use of ELSE is optional. We've included it in the preceding syntax to show where it goes if you decide to include it.

If *condition1* is true, QBasic

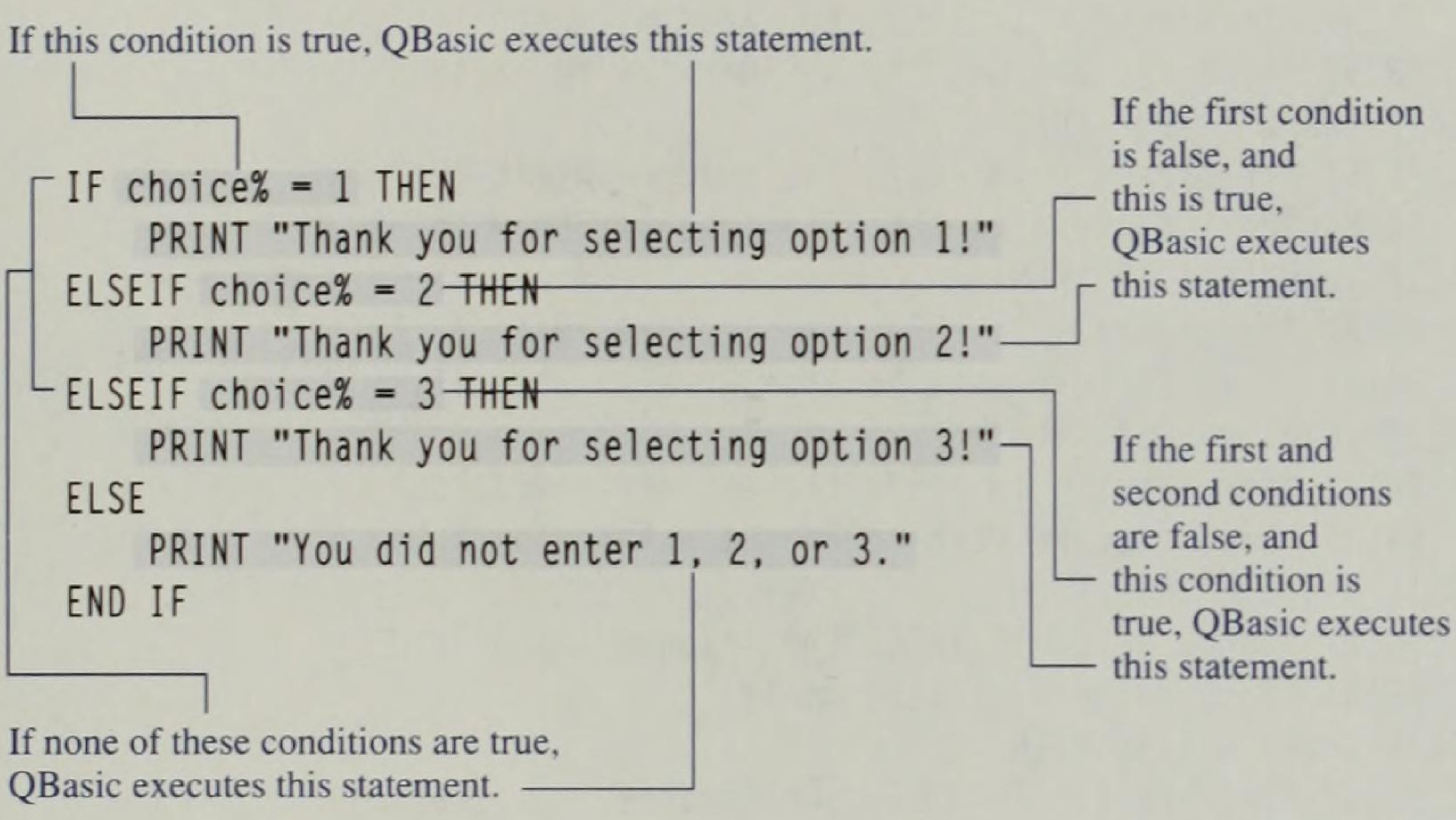
1. Executes the statements on the following lines until it encounters the first ELSEIF
2. Jumps down to END IF
3. Continues executing the program

If *condition1* is false, QBasic jumps down to the first ELSEIF and evaluates *condition2*. If *condition2* is true, QBasic

1. Executes the statements on the following lines until it encounters the next ELSEIF or, if you included one, the ELSE clause
2. Jumps down to END IF and continues executing the program

If the condition associated with an ELSEIF statement is false, QBasic jumps down to the next ELSEIF statement and evaluates its condition. QBasic continues this process until an ELSEIF statement evaluates to true or until it encounters an ELSE statement.

Here's an example:



```
IF choice% = 1 THEN
    PRINT "Thank you for selecting option 1!"
ELSEIF choice% = 2 THEN
    PRINT "Thank you for selecting option 2!"
ELSEIF choice% = 3 THEN
    PRINT "Thank you for selecting option 3!"
ELSE
    PRINT "You did not enter 1, 2, or 3."
END IF
```

If this condition is true, QBasic executes this statement.

If the first condition is false, and this is true, QBasic executes this statement.

If the first and second conditions are false, and this condition is true, QBasic executes this statement.

If none of these conditions are true, QBasic executes this statement.

Notice how using the ELSEIF keyword allows you to specify different conditions and different courses of action for QBasic to take.

Practice: Working with ELSEIF

1. Type in the ELSEIF.BAS program (Figure 5-6).

```
' ELSEIF.BAS
' This program demonstrates the use of ELSEIF.

CLS

PRINT "Welcome to Motion Picture Trivia!"
PRINT "In what year did the film Ben Hur win"
PRINT "the Academy Award for Best Picture?"
PRINT
INPUT "Please enter a year from 1950 through 1959:  ", year%
PRINT
```

FIGURE 5-6. *(continued)*
ELSEIF.BAS. A motion-picture trivia game that uses ELSEIF.

FIGURE 5-6. *continued*

```
IF year% = 1950 THEN
    PRINT "Incorrect.  In 1950, the Academy Award for"
    PRINT "Best Picture went to All About Eve."
ELSEIF year% = 1951 THEN
    PRINT "Incorrect.  In 1951, the Academy Award for"
    PRINT "Best Picture went to An American in Paris."
ELSEIF year% = 1952 THEN
    PRINT "Incorrect.  In 1952, the Academy Award for"
    PRINT "Best Picture went to The Greatest Show on Earth."
ELSEIF year% = 1953 THEN
    PRINT "Incorrect.  In 1953, the Academy Award for"
    PRINT "Best Picture went to From Here to Eternity."
ELSEIF year% = 1954 THEN
    PRINT "Incorrect.  In 1954, the Academy Award for"
    PRINT "Best Picture went to On the Waterfront."
ELSEIF year% = 1955 THEN
    PRINT "Incorrect.  In 1955, the Academy Award for"
    PRINT "Best Picture went to Marty."
ELSEIF year% = 1956 THEN
    PRINT "Incorrect.  In 1956, the Academy Award for"
    PRINT "Best Picture went to Around the World in 80 Days."
ELSEIF year% = 1957 THEN
    PRINT "Incorrect.  In 1957, the Academy Award for"
    PRINT "Best Picture went to The Bridge on the River Kwai."
ELSEIF year% = 1958 THEN
    PRINT "Incorrect.  In 1958, the Academy Award for"
    PRINT "Best Picture went to Gigi."
ELSEIF year% = 1959 THEN
    PRINT "Correct!  Ben Hur, directed by William Wyler,"
    PRINT "won 11 Academy Awards in 1959, including"
    PRINT "Best Picture."
ELSE
    PRINT "You did not enter a number from 1950 through 1959."
    PRINT "Please run the program again and enter an"
    PRINT "appropriate value."
END IF
```

2. Run the program and enter a value of *1959*, which is the correct answer to the question. Your output screen will look like this:

```
Welcome to Motion Picture Trivia!
In what year did the film Ben Hur win
the Academy Award for Best Picture?

Please enter a year from 1950 through 1959:  1959

Correct!  Ben Hur, directed by William Wyler,
won 11 Academy Awards in 1959, including
Best Picture.
```

3. Run the program again, this time entering a value other than 1959. Your output screen will look something like this:

```
Welcome to Motion Picture Trivia!
In what year did the film Ben Hur win
the Academy Award for Best Picture?

Please enter a year from 1950 through 1959:  1950

Incorrect.  In 1950, the Academy Award for
Best Picture went to All About Eve.
```

4. Run the program one more time, this time entering a value that is not in the range 1950 through 1959. Your output screen will look something like this:

```
Welcome to Motion Picture Trivia!
In what year did the film Ben Hur win
the Academy Award for Best Picture?

Please enter a year from 1950 through 1959:  1596

You did not enter a number from 1950 through 1959.
Please run the program again and enter an
appropriate value.
```

THE SELECT CASE STATEMENT

Another tool that allows you to work with conditional statements is the SELECT CASE statement. The SELECT CASE statement is similar in function to the IF statement. In fact, in many cases you can use either an IF statement or a SELECT CASE statement to perform the same job. However, the differences between the two make each better suited to a particular set of circumstances.

The CASE and END SELECT Keywords

Just as the IF statement needs other keywords in order to do its job properly, the SELECT CASE statement needs to be used with other QBasic keywords—namely CASE and END SELECT—if it is to do its work.

Here is the syntax for a SELECT CASE statement (related statements are indented for clarity):

```
SELECT CASE variable
    CASE value1
        statements to be executed if value1 matches variable
    CASE value2
        statements to be executed if value2 matches variable
    CASE value3
        statements to be executed if value3 matches variable
    ⋮
END SELECT
```

Unlike an IF statement, a complete SELECT CASE statement can't be specified on a single line. You must use the block syntax shown.

variable can be a numeric or a string variable. You can think of the variable following SELECT CASE as the "gateway" to the individual CASE clauses below it—QBasic uses the value of *variable* to determine which CASE clause to use.

The CASE clauses are separate conditional statements. The *value* of each is related to *variable*. There is no limit to the number of individual CASE clauses you can put between SELECT CASE and END SELECT.

Each CASE clause is followed by one or more QBasic statements that QBasic executes if the value in the preceding CASE clause matches the value of the variable following SELECT CASE. Here's an example:

If the value of this variable matches
this value, QBasic
executes this statement.

```
SELECT CASE guess%
    CASE 1
        PRINT "Sorry, 1 is not the number."
    CASE 2
        PRINT "Sorry, 2 is not the number."
    CASE 3
        PRINT "That's right!  3 is the right number!"
    CASE 4
        PRINT "Sorry, 4 is not the number."
END SELECT
```

When QBasic encounters a SELECT CASE statement, it takes note of the value of the variable (in this example, the value of *guess%*) and then examines the value specified in the first CASE clause. If that value matches the value of the variable, QBasic

1. Executes the statements following that CASE until it encounters the next CASE or the END SELECT statement
2. Jumps down to the statement following END SELECT and continues executing the program

If the value in the first CASE clause does not match the value of the variable, QBasic checks the value of each CASE clause until it finds a match.

If none of the CASE clauses contain a value matching the value of the variable following SELECT CASE, QBasic jumps down to the statement following END SELECT and continues executing the program.

Practice: Working with the SELECT CASE statement

1. Type in the SELECT-1.BAS program (Figure 5-7).

```
' SELECT-1.BAS
' This program asks the user for his or her name, prints a menu,
'   and asks the user to choose a menu item.  The program then
'   displays a message based on which menu item the user chose.

CLS

INPUT "Please enter your name:  ", userName$
PRINT
PRINT "Congratulations, "; userName$; "!  You've won the final"
PRINT "round of Go Ahead--Make My Deal!!  Please choose"
PRINT "which prize you want:"
PRINT
PRINT , "1.  Door #1"
PRINT , "2.  What's behind the curtain"
PRINT , "3.  The Big Pink Box"
PRINT
INPUT "Please enter 1, 2, or 3:  ", menuNum%
PRINT

SELECT CASE menuNum%
    CASE 1
        PRINT "** It's a new car!!! **"
        PRINT "Yes, it's the new Land Yacht 2000,"
        PRINT "complete with Guzzle-O-Matic!"
    CASE 2
        PRINT "** 500 pogo sticks!!! **"
        PRINT "Yes, you and your entire family can hop till you"
        PRINT "drop with the latest in pogoing fun!"
    CASE 3
        PRINT "** A lifetime supply of pickled herring!!! **"
        PRINT "Yes, twelve thousand jars of Smelz-Good"
        PRINT "pickled herring delivered to your front door!"
END SELECT

PRINT
PRINT "Thanks for playing!"
```

FIGURE 5-7.
SELECT-1.BAS: A program demonstrating the SELECT CASE statement.

2. Run the program to see output something like this:

```
Please enter your name:  Torvald

Congratulations, Torvald!  You've won the final
round of Go Ahead--Make My Deal!!  Please choose
which prize you want:

              1.  Door #1
              2.  What's behind the curtain
              3.  The Big Pink Box

Please enter 1, 2, or 3:  2

** 500 pogo sticks!!! **
Yes, you and your entire family can hop till you
drop with the latest in pogoing fun!

Thanks for playing!
```

   After you entered a value for *menuNum%*, QBasic used it to check the value in each CASE clause. Because the user entered the value 2 here, QBasic found a match in the second CASE clause and executed the statements associated with that CASE clause. Then, because it had found a match, QBasic jumped over the remaining CASE clause and executed the remainder of the program.

3. Run the program again, but this time use a value other than 1, 2, or 3. Your output screen will look something like this:

```
Please enter your name:  Dexter

Congratulations, Dexter!  You've won the final
round of Go Ahead--Make My Deal!!  Please choose
which prize you want:

              1.  Door #1
              2.  What's behind the curtain
              3.  The Big Pink Box

Please enter 1, 2, or 3:  5


Thanks for playing!
```

This time QBasic didn't display a prize-winning message because it couldn't find a matching value in any of the CASE clauses.

Using CASE ELSE

Recall from earlier in the chapter that, when used in an IF statement, the ELSE clause lets you specify one or more actions to be executed when conditional expressions evaluate to false. You use CASE ELSE to do the same job with a SELECT CASE statement.

The syntax for a SELECT CASE statement using CASE ELSE is

```
SELECT CASE variable
    CASE value
        statements to be executed if value matches variable
    :
    CASE ELSE
        statements to be executed if no values match variable
END SELECT
```

Practice: Using a CASE ELSE statement

1. Type in the SELECT-2.BAS program (Figure 5-8).

```
' SELECT-2.BAS
' This program demonstrates the use of CASE ELSE.

CLS

PRINT "Welcome to Golden-Age Television Trivia!"
PRINT "In what year did the top-rated show"
PRINT "I Love Lucy first appear on television?"
PRINT
INPUT "Please enter a year from 1950 through 1959:  ", year%
PRINT

SELECT CASE year%
    CASE 1950
        PRINT "Incorrect.  In 1950, the popular show"
        PRINT "Your Show of Shows made its television debut."
```

FIGURE 5-8. *(continued)*
SELECT-2.BAS: A television trivia game that uses CASE ELSE.

FIGURE 5-8. *continued*

```
    CASE 1951
        PRINT "Correct!  I Love Lucy first aired in October 1951."
        PRINT "From 1952 through 1954 it was the most popular"
        PRINT "television show in the United States."
    CASE 1952
        PRINT "Incorrect.  In 1952, the popular show"
        PRINT "Dragnet made its television debut."
    CASE 1953
        PRINT "Incorrect.  In 1953, the popular show"
        PRINT "Name That Tune made its television debut."
    CASE 1954
        PRINT "Incorrect.  In 1954, the popular show"
        PRINT "Lassie made its television debut."
    CASE 1955
        PRINT "Incorrect.  In 1955, the popular show"
        PRINT "Gunsmoke made its television debut."
    CASE 1956
        PRINT "Incorrect.  In 1956, the popular show"
        PRINT "Playhouse 90 made its television debut."
    CASE 1957
        PRINT "Incorrect.  In 1957, the popular show"
        PRINT "Leave It to Beaver made its television debut."
    CASE 1958
        PRINT "Incorrect.  In 1958, the popular show"
        PRINT "Peter Gunn made its television debut."
    CASE 1959
        PRINT "Incorrect.  In 1959, the popular show"
        PRINT "The Twilight Zone made its television debut."
    CASE ELSE
        PRINT "You did not enter a number from 1950 through 1959."
        PRINT "Please run the program again and enter an"
        PRINT "appropriate value."
END SELECT
```

2. Run the program and enter the value *1951*—the correct answer to the trivia question. Your output screen will look like this:

```
Welcome to Golden-Age Television Trivia!
In what year did the top-rated show
I Love Lucy first appear on television?

Please enter a year from 1950 through 1959:  1951

Correct!  I Love Lucy first aired in October 1951.
From 1952 through 1954 it was the most popular
television show in the United States.
```

3. Now run the program again, but this time enter a value outside the proper range. Your output screen will look something like this:

```
Welcome to Golden-Age Television Trivia!
In what year did the top-rated show
I Love Lucy first appear on television?

Please enter a year from 1950 through 1959:  1598

You did not enter a number from 1950 through 1959.
Please run the program again and enter an
appropriate value.
```

   Because QBasic couldn't find a matching CASE clause, it executed the statements following CASE ELSE. Using CASE ELSE is a convenient means of handling unforeseen difficulties in your SELECT CASE statements.

Using IS with CASE

The IS keyword lets you use a conditional *expression* (instead of only a single numeric or string *value*) in any CASE clause. QBasic executes the statements following that CASE clause only if the condition is true. Here's the syntax for a CASE clause using IS:

```
SELECT CASE variable
    CASE IS condition
        statements to be executed if condition is true
    ⋮
END SELECT
```

This syntax is identical to the standard SELECT CASE statement, except for the addition of "IS *condition*" to the CASE clause. When it evaluates the condition, QBasic uses the value of the SELECT CASE variable as the basis of the comparison.

Here's an example:

If the value of *userNum%* is less than or equal to 4, QBasic executes this statement.

If the value of *userNum%* is 5, QBasic executes this statement.

```
SELECT CASE userNum%
    CASE IS <= 4
        PRINT "The number you entered was 4 or less."
    CASE 5
        PRINT "The number you entered was 5."
    CASE IS >= 6
        PRINT "The number you entered was 6 or more."
END SELECT
```

If the value of *userNum%* is greater than or equal to 6, QBasic executes this statement.

QBasic compares the value entered by the user with the conditional portions of the CASE clauses. In this example, if the user entered a value of 4 or less, the condition in the first CASE clause would be true, so QBasic would execute the associated PRINT statement. (The second CASE clause simply uses a value. You can mix values and IS conditional expressions.)

Practice: Working with SELECT CASE and IS

1. Type in the SELECT-3.BAS program (Figure 5-9).
2. Run the program and enter a value. Your output screen will look something like this:

```
How many cups of coffee will you drink today? 3

A moderate level.
```

```
' SELECT-3.BAS
' This program analyzes the user's daily coffee consumption.

CLS

INPUT "How many cups of coffee will you drink today?  ", cupsCoffee%
PRINT

SELECT CASE cupsCoffee%
    CASE 0
        PRINT "Don't you LIKE my coffee??"
    CASE IS <= 3          ' 1 to 3 cups a day
        PRINT "A moderate level."
    CASE IS <= 7          ' 4 to 7 cups a day
        PRINT "I see decaf in your future..."
    CASE IS >= 8          ' 8 or more cups a day
        PRINT "Caffeine overload!"
END SELECT
```

FIGURE 5-9.
SELECT-3.BAS: A coffee-consumption "analysis" program that uses IS.

Press a key to return to the View window. In the program, notice how the conditions are set up. If you entered the value *3* when you ran the program, as we did, you'll see that it caused the second CASE clause to be true. Notice also, however, that technically the value 3 would cause the third CASE clause to be true as well, and you might be wondering why QBasic didn't execute the associated PRINT statement. Remember: As soon as QBasic finds a CASE clause with a true condition, it jumps over the remaining CASE clauses without even considering them and continues executing your program.

Using TO with SELECT CASE

To specify an inclusive range of valid values in a CASE clause, use the TO keyword. The syntax for using TO in a CASE clause is as follows:

```
SELECT CASE variable
    CASE value1 TO value2
        statements to be executed if CASE is true
    :
END SELECT
```

The two values in the CASE clause can be two numeric values or two text strings, each in double quotation marks.

Using numeric values with the TO keyword

Here's an example of using two numeric values with the TO keyword in a CASE clause:

If the value of *userNum%* is 1 through 5, inclusive, QBasic executes this statement.

```
SELECT CASE userNum%
    CASE 1 TO 5
        PRINT "The number you entered was from 1 through 5."
    CASE 6 TO 10
        PRINT "The number you entered was from 6 through 10."
END SELECT
```

If the value of *userNum%* is 6 through 10, inclusive, QBasic executes this statement.

Note that the lesser of the two values is to the left of the TO keyword. You must list the values in this order; if you don't, QBasic will misinterpret the condition and will always evaluate the expression as false.

This is also true for negative values. Here's an example using a negative number in the correct position:

```
CASE -12 TO 3
```

Because negative 12 is less than positive 3, you must put the negative 12 to the left of the TO keyword.

Using text strings with the TO keyword

You can also use a range of text strings in a CASE clause if you use the TO keyword. Here's an example:

```
SELECT CASE word$
    CASE "a" TO "m"
        PRINT "The word you entered was in the range a to m."
    CASE "m" TO "z"
        PRINT "The word you entered was in the range m to z."
END SELECT
```

At first glance, a range of text strings might not seem as obvious as a range of numbers, but both work in a similar way.

Recall from a previous example that QBasic treats each character separately. For example, as far as QBasic is concerned, the letters *a* and *A* are different.

The reason for this lies in how your computer deals with information. Although you type uppercase and lowercase letters on your keyboard, and although your computer can display uppercase and lowercase letters on your screen, your computer cannot work with letters directly. The microchips inside your computer can work only with numbers. (And you thought computers were smart....) When you press a character key on your keyboard, your computer temporarily translates that character into a number it can work with. Then, when it needs to do something with that number in its character form, such as display it on your screen, QBasic changes the number back to the character you originally typed in.

Relax—you don't need to learn how this numbering business works, but you do need to know that it happens if you intend to understand how QBasic works with individual characters or strings of characters. Even though you deal with the characters or strings of characters themselves, your computer and QBasic simply consider them numbers.

In the preceding example, notice the overlap between the two CASE clauses. In the first CASE clause the range is from *a* through *m*, and in the second CASE clause the range is from *m* through *z*. Why the overlap? The following practice session will demonstrate.

Practice: Working with a range of text strings in a CASE clause

1. Type in the TO-1.BAS program (Figure 5-10).

```
' TO-1.BAS
' This program demonstrates the TO keyword in a CASE clause.

CLS

INPUT "Please enter a name:  ", name$
PRINT

SELECT CASE name$
    CASE "a" TO "m"
        PRINT "The name is in the range a to m."
    CASE "m" TO "z"
        PRINT "The name is in the range m to z."
    CASE "A" TO "M"
        PRINT "The name is in the range A to M."
    CASE "M" TO "Z"
        PRINT "The name is in the range M to Z."
END SELECT
```

FIGURE 5-10.
TO-1.BAS: A text-comparison program that uses TO.

2. Run the program and enter the name *billy*. Your output screen will look like this:

   ```
   Please enter a name:  billy

   The name is in the range a to m.
   ```

   Because the first letter of *billy* is a lowercase *b*, it caused the first CASE statement to be true because a lowercase *b* falls within the range of *a* to *m*.

3. Run the program again, and enter only the letter *m*. Your output screen will look like this:

   ```
   Please enter a name:  m

   The name is in the range a to m.
   ```

   Again, because the lowercase letter *m* caused the first CASE to be true, QBasic executed the statement following the first CASE.

4. Run the program again, and enter the name *mildred.* Your output screen will look like this:

```
Please enter a name:  mildred

The name is in the range m to z.
```

Notice that entering the word *mildred* caused the second CASE to be true, but that in the previous example entering only the letter *m* caused the first CASE to be true. Why did this happen?

Recall that inside a computer, all letters are treated as numbers. In a sense, then, you can think of a string of characters as a series of numbers. In the preceding example, the word *mildred* had a higher ''value'' than the letter *m,* just as the number 15 has a higher value than the number 1. Even though both numbers—1 and 15—begin with a 1, the rules of numbers and mathematics tell you that 15 has a greater value than 1.

This rule loosely applies to strings of characters, but there is a difference. In the example above, QBasic didn't consider the entire word *mildred* when it examined the CASE clauses. It needed merely to examine the first two letters—*mi.* Because the letter combination *mi* has a higher ''value'' than simply the letter *m,* and because the single letter *m* is at the top of the range of *a* to *m,* QBasic needed to look no further—it knew the first condition was not true.

QBasic did, however, find a match in the second condition. Although the string *mildred* is longer than the single-letter match QBasic was looking for, the first letter of *mildred* does fall within the range of *m* to *z,* so QBasic determined that the second condition was true.

Practice: More work with a range of text strings

1. Modify the TO-1.BAS program so that it matches the TO-2.BAS program (Figure 5-11).

```
' TO-2.BAS
' This program demonstrates the TO keyword in a CASE clause.

CLS
```

FIGURE 5-11. *(continued)*
TO-2.BAS: A modified version of TO-1.BAS.

FIGURE 5-11. *continued*

```
INPUT "Please enter a name:  ", name$
PRINT

SELECT CASE name$
    CASE "a" TO "mz"
        PRINT "The name is in the range a to m."
    CASE "n" TO "z"
        PRINT "The name is in the range n to z."
    CASE "A" TO "Mz"
        PRINT "The name is in the range A to M."
    CASE "N" TO "Z"
        PRINT "The name is in the range N to Z."
END SELECT
```

TO-2.BAS is identical to TO-1.BAS except for some changes to the conditions—note especially the first and third CASE clauses.

2. Run the program, and enter a capital *M.* (We'll use capital letters this time for a little variety.) Your output screen will look like this:

```
Please enter a name:  M

The name is in the range A to M.
```

3. Now run the program again, and enter the name *Mildred.* Your output screen will look like this:

```
Please enter a name:  Mildred

The name is in the range A to M.
```

This time, the condition in the third CASE clause is true because *Mz* has a greater "value" than the first two letters of *Mildred*, the name you typed in.

Using Multiple Conditions with CASE

So far, the condition you've used in each CASE clause has been a single value, condition, or range of values. To specify multiple conditions within a single CASE clause, simply separate the conditions with commas. You'll find an example on the next page.

If any of these values matches the value of *number%*, QBasic executes this statement.

```
SELECT CASE number%
    CASE 1, 3, 5, 7, 9
        PRINT "The number you entered was odd."
    CASE 2, 4, 6, 8, 10
        PRINT "The number you entered was even."
END SELECT
```

If any of these values matches the value of *number%*, QBasic executes this statement.

Using a comma within a CASE clause is similar to using the OR operator in an IF statement: If one of the items causes that CASE to be true, QBasic executes the associated statements.

Items needn't be of the same type: You can mix values, conditions, and ranges of values. For example, the following CASE clause is valid:

```
CASE 5, IS <> 6, 20 TO 30
```

Practice: Working with commas and CASE

Type in the TO-3.BAS program (Figure 5-12) and run it. As you can see, using commas to specify multiple items allows you greater flexibility in your SELECT CASE statements.

```
' TO-3.BAS
' This program demonstrates the use of commas with CASE.

CLS

PRINT "Enter a month, and I will tell you how many"
PRINT "U.S. holidays there are in that month."
INPUT "Please enter a number from 1 through 12:  ", month%
PRINT

SELECT CASE month%
    CASE 8
        PRINT "There are no U.S. holidays in that month."
    CASE 3, 4, 7
        PRINT "There is 1 U.S. holiday in that month."
```

FIGURE 5-12. *(continued)*
TO-3.BAS: A program demonstrating the use of multiple conditions within CASE clauses.

FIGURE 5-12. *continued*

```
    CASE 6
       PRINT "There are 2 U.S. holidays in that month."
    CASE 1, 2, 9 TO 11
       PRINT "There are 3 U.S. holidays in that month."
    CASE 12
       PRINT "There are 4 U.S. holidays in that month."
    CASE 5
       PRINT "There are 5 U.S. holidays in that month."
 END SELECT
```

WHAT KIND OF CONDITIONAL STATEMENT SHOULD YOU USE?

Both the IF and the SELECT CASE statements have plenty of options and many different configurations. When you want to include one or more conditional statements in your program, the answers to the following questions should help you choose the most appropriate statement for your needs.

- **Which statement are you most comfortable using?** Through the help of associated keywords such as ELSE and ELSEIF with an IF statement, and CASE and CASE ELSE with a SELECT CASE statement, you can get the same result using either statement. Consider using the statement you're most comfortable with.
- **How many conditions does your statement have?** SELECT CASE requires a minimum of three separate instructions to set up even a single condition! So, if you have only one condition, a simple IF statement is your best bet. If your statement has two or more conditions, IF and SELECT CASE are equally easy to use.

  If you have many conditions, SELECT CASE is a good choice. It is visually less cluttered and therefore easier to read and comprehend than an IF statement. A SELECT CASE statement is also a good candidate for situations in which you're looking for a number of specific values that don't fit neatly into a range.

SUMMARY

Conditional statements allow your programs to be much more flexible and “smart.” By using conditional statements, you can control how your program runs, based on information typed in by the user or on information changed by the program as it runs. You've just taken another major step on your journey toward becoming a QBasic programmer—you'll use conditional statements as an integral part of most programs you write.

QUESTIONS AND EXERCISES

1. What are the differences between a numeric expression and a conditional expression?
2. Which of the following are *not* conditional operators?

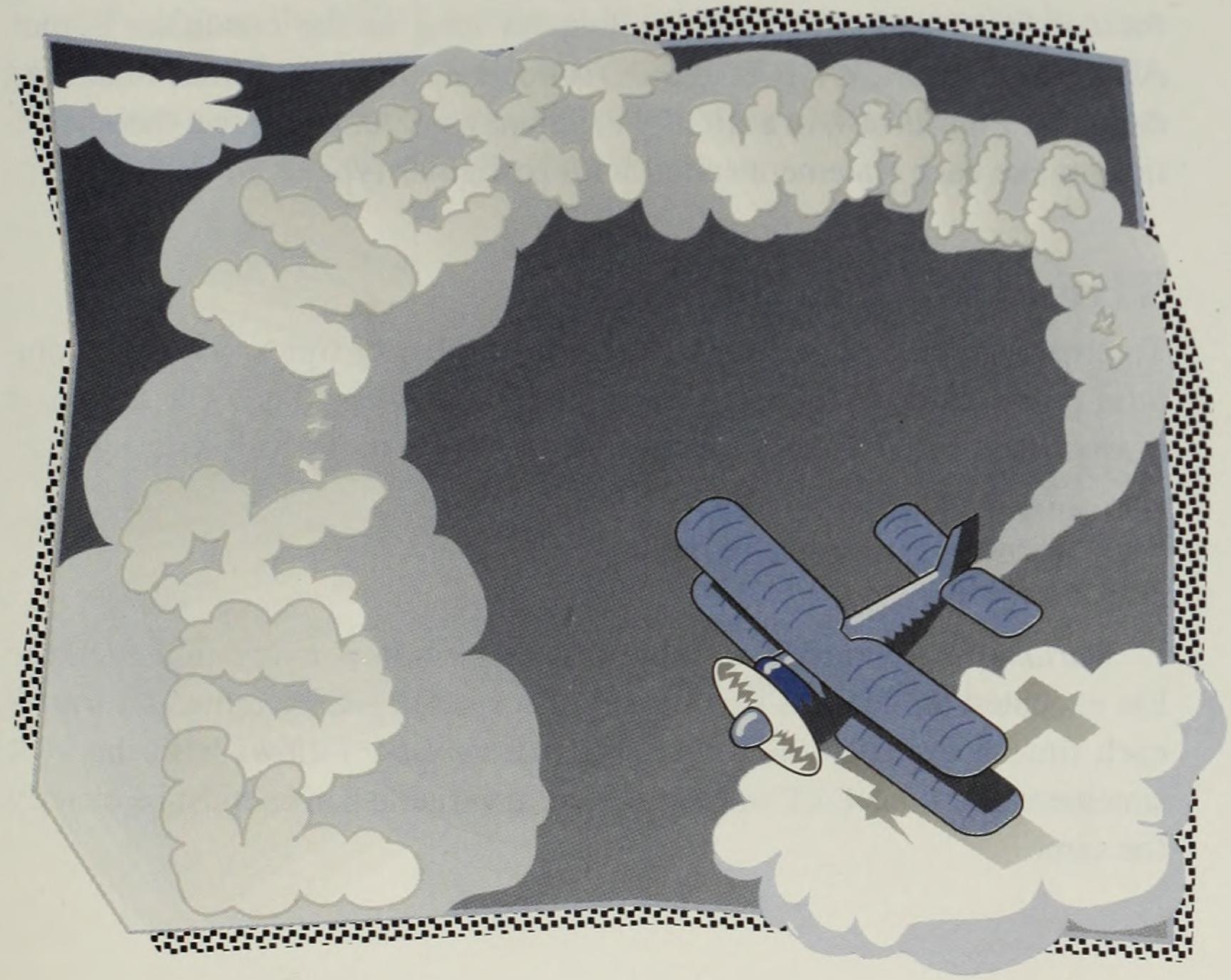
   a. <=
   b. ><
   c. ==
   d. <
   e. >=
   f. ^
   g. <>
   h. >
   i. =<
   j. /
   k. <<
   l. =

3. True or False: You must always use the THEN keyword on the same line as the IF keyword.
4. What is the difference between the AND operator and the OR operator?
5. What does the ELSE keyword allow you to do? How does it differ from the THEN keyword?
6. How does the ELSEIF keyword work? What other QBasic keyword must you use with ELSEIF?
7. Write a program that asks the user a yes-or-no question. Instruct the user to enter a *Y* for yes or an *N* for no. Include a set of statements that will be executed if the user enters *Y* and a separate set of statements that will be executed if the user enters *N*. Include a third set of statements that will be executed if the user does not enter a proper response. (Hint: Remember to check for uppercase and lowercase letters.)
8. How does the SELECT CASE statement work?
9. What does the CASE ELSE clause do?

10. In a CASE statement, what do the IS and TO keywords do?
11. Write a program that offers to describe three items to the user. Have the program display a numbered list of the items, and then have the program ask the user to enter the number corresponding to the item to be described. Use a SELECT CASE statement to display appropriate information, and also include statements that will be executed if the user does not enter an appropriate response.

CHAPTER 6

Working with QBasic Loops



You've now learned most of the fundamentals of the QBasic language. In this chapter, you'll learn how to use one of QBasic's most powerful tools: the *loop*. By using loops in QBasic, you can have the computer perform certain repetitive tasks in a fraction of the time it would take to do the job yourself.

INTRODUCTION TO QBASIC LOOPS

A *loop* is simply one or more QBasic statements that you direct QBasic to repeat. Loops can repeat in two ways:

- A specific number of times
- Until a certain condition is met

In QBasic, three statements—FOR, WHILE, and DO—allow you to add loops to your program. In the FOR statement, you specify the number of times you want a loop to be executed. In the WHILE and DO statements, you specify a condition, and QBasic is responsible for evaluating the condition and executing the loop as long as the condition is met. Although the FOR, WHILE, and DO statements perform similar functions, each performs its task in a slightly different way. As you'll see, their variations make each statement suited for a particular type of work.

THE FOR STATEMENT

To create a loop that executes a specific number of times (the most common type of loop in QBasic), use the FOR statement. The FOR statement always ends with the NEXT statement, as shown in the following syntax:

```
FOR variable = start TO end
     statements to be repeated
NEXT variable
```

variable is a numeric variable that reflects how many times QBasic has executed the loop. It's a counter of sorts. QBasic increments *variable* each time it executes the loop. Note that *variable* follows both the FOR statement and the NEXT statement. These variable names must be exactly the same.

start is the numeric value, either a number or a numeric expression, at which you want QBasic to start counting.

end is a numeric value, either a number or a numeric expression, that tells QBasic how high it should count, that is, how many times it should repeat the loop. When you assign values to *start* and *end*, keep the following hints in mind:

- You can use both positive and negative values for *start* and for *end*.
- You can use both integers and floating-point values for *start* and for *end*.
- The value of *start* must be less than or equal to the value of *end*.
- The value of *start* doesn't need to be 1.

The statements between the FOR statement and the NEXT statement are the statements that QBasic executes the specified number of times. There is no limit to the number of statements you can use, and the statements don't need to be of the same type.

QBasic uses the NEXT statement to count how many times it has executed the statements between the FOR statement and the NEXT statement. Here's an example of a FOR statement that prints a message five times, incrementing the variable *i%* each time it does so:

```
FOR i% = 1 TO 5
   PRINT "I am in a loop."
NEXT i%
```

Each time it completes the loop, QBasic jumps back up to the FOR statement to compare the value of *i%* with the value of *end* (which is 5). As soon as *i%* exceeds *end*, QBasic jumps to the statement following NEXT and continues executing the program.

Practice: Working with a FOR loop

1. Type in the FOR-1.BAS program (Figure 6-1).

```
' FOR-1.BAS
' This program demonstrates the FOR loop.

CLS

INPUT "Please enter a number from 2 through 5:  ", times%
PRINT

FOR i% = 1 TO times%
    PRINT "These lines will print"; times%; "times."
    PRINT "This is time"; i%
    PRINT
NEXT i%
```

FIGURE 6-1.
FOR-1.BAS: A simple FOR loop.

2. Run the program. Your output screen should look something like this:

```
Please enter a number from 2 through 5:  4

These lines will print 4 times.
This is time 1

These lines will print 4 times.
This is time 2

These lines will print 4 times.
This is time 3

These lines will print 4 times.
This is time 4
```

Practice: When **start** *is greater than* **end**

1. Type in the FOR-2.BAS program (Figure 6-2). Note that *start* is greater than *end.*

```
' FOR-2.BAS
' This program demonstrates a FOR loop that never loops.

CLS

FOR i% = 6 TO 5
   PRINT "The current value of i% is"; i%
NEXT i%
```

FIGURE 6-2.
FOR-2.BAS: A FOR loop that never loops.

2. Run the program. Except for the message *Press any key to continue*, your output screen will be blank. QBasic saw the counter at *6*, a value greater than *end*, and therefore considered its task complete.

Why Use *i%*?

In most of the FOR loops in this book, you'll notice that *i%* is the loop variable. Why is this?

Before Basic was invented, many programmers used a language called FORTRAN. In FORTRAN, the first letter of a variable name specified its type. For instance, any variable name that started with a letter from *I* through *N* denoted an integer variable, much as a percent sign (%) appended to a QBasic variable name denotes an integer variable.

Most FOR loop counters are integers. Thus, to save typing time, a FORTRAN programmer usually used the variable *i* as the loop counter. Programmers needing to nest two or more loops (see ''Nesting FOR Loops'' later in this chapter) would use *j*, then *k*, and so on. This tradition has been adopted by many Basic programmers.

Must you use *i%*, *j%*, and *k%*? No. You can use any valid QBasic variable name—for example, *count%*, *numLines%*, or *dayOfMonth%*.

Practice: Using equal* start *and* end *values

1. Change the FOR statement in the FOR-2.BAS program so that *start* and *end* have the same value:

```
FOR i% = 5 TO 5
```

2. Run the program. Your output screen will look like this:

```
The current value of i% is 5
```

This time, QBasic executed the body of the FOR statement once. Because the *start* and *end* values are the same, QBasic assumed that this was its last ''lap'' and executed the PRINT statement one time before moving on.

Looping and the COLOR Statement

You'll learn more about the COLOR statement in Chapter 11, but for now just sit back and enjoy this demonstration, which uses both the COLOR statement and a FOR loop within a program.

Practice: Changing foreground colors with a FOR loop

1. Type in the FOR-3.BAS program (Figure 6-3).

```
' FOR-3.BAS
' This program uses a FOR loop and the COLOR statement
'   to demonstrate the various foreground colors.

CLS

FOR i% = 1 TO 15         ' use colors 1 through 15
    COLOR i%             ' use value of i% for color
    PRINT "This line is printed using color"; i%
NEXT i%
```

FIGURE 6-3.
FOR-3.BAS: Using the COLOR statement in a FOR loop.

NOTE: You can run this program even if you don't have a color monitor attached to your computer. Your experience won't be as colorful, but it will be enlightening all the same.

The COLOR Statement

The COLOR statement lets you specify the foreground (text) and background colors for your output screen. Here's the syntax for the COLOR statement:

```
COLOR [foreground][, background]
```

Both *foreground* and *background* are numbers that represent colors:

Value	***Resulting color***
0	Black (default background)
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White (default foreground)
8	Gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	Bright white

For example, the statement

```
COLOR 14, 1
```

causes QBasic to display yellow text on a blue background.

2. Run the program. Your output screen will look like this, except that each line will be a different color:

```
This line is printed using color 1
This line is printed using color 2
This line is printed using color 3
This line is printed using color 4
This line is printed using color 5
This line is printed using color 6
This line is printed using color 7
This line is printed using color 8
This line is printed using color 9
This line is printed using color 10
This line is printed using color 11
This line is printed using color 12
This line is printed using color 13
This line is printed using color 14
This line is printed using color 15
```

Practice: Changing foreground and background colors with a FOR loop

1. Type in the FOR-4.BAS program (Figure 6-4).

```
' FOR-4.BAS
' This program uses a FOR loop and the COLOR statement
'   to change both the foreground and background colors.

CLS

FOR i% = 1 TO 15         ' use colors 1 through 15
    COLOR i%, i% - 1
    PRINT "Current foreground color is"; i%;
    PRINT "; current background color is"; i% - 1
NEXT i%
```

FIGURE 6-4.
FOR-4.BAS: Changing both foreground and background colors with a FOR loop.

2. Run the program. Your output screen will look like this but will be in color:

```
Current foreground color is 1; current background color is 0
Current foreground color is 2; current background color is 1
Current foreground color is 3; current background color is 2
Current foreground color is 4; current background color is 3
Current foreground color is 5; current background color is 4
Current foreground color is 6; current background color is 5
Current foreground color is 7; current background color is 6
Current foreground color is 8; current background color is 7
Current foreground color is 9; current background color is 8
Current foreground color is 10; current background color is 9
Current foreground color is 11; current background color is 10
Current foreground color is 12; current background color is 11
Current foreground color is 13; current background color is 12
Current foreground color is 14; current background color is 13
Current foreground color is 15; current background color is 14
```

3. Press a key to return to the View window, and then look at the COLOR statement in the FOR loop. Notice the values that set the foreground and background colors. The foreground color is set to the current value of *i%*, and the background color is set to the value of *i%* minus one. Using a technique such as this prevents setting the foreground color and background color to the same value—if the foreground and background values are the same, you won't be able to read the text.

Looping and the SOUND Statement

Another fun QBasic statement is SOUND. (Again, you'll be hearing more about SOUND in Chapter 11.) In the following practice, the FOR statement combined with the SOUND statement provides audible proof of the power of loops.

The SOUND Statement

The SOUND statement does exactly what the name implies—it causes your computer to create a sound. Here's the syntax for the SOUND statement:

```
SOUND frequency, duration
```

The *frequency* argument is an integer from 37 through 32,767 that indicates the frequency of the sound in cycles per second, or hertz. The *duration* argument is an integer or floating-point number from 0 through 65,535 that indicates how long the sound should last. (The value 18.2 equals one second.) For example, the following statement causes QBasic to play a sound of 500 hertz for one-half second:

```
SOUND 500, 9.1
```

Practice: Using the FOR and SOUND statements

1. Type in the FOR-5.BAS program (Figure 6-5).

```
' FOR-5.BAS
' This program uses the SOUND statement inside a
'   FOR loop to generate sound effects.

CLS

PRINT , "SOUND-EFFECTS GENERATOR"
PRINT , "Please choose the sound you'd like to hear"
PRINT , "from the following menu:"
PRINT
PRINT , "1.  Ray gun"
PRINT , "2.  European siren"
PRINT , "3.  Takeoff!"
PRINT
```

FIGURE 6-5.
FOR-5.BAS: A sound-effects generator.

(continued)

FIGURE 6-5. *continued*

```
INPUT "              Please enter 1, 2, or 3:  ", choice%
PRINT
SELECT CASE choice%
    CASE 1                      ' ray gun
        FOR i% = 1 TO 50
            SOUND 850, .3
            SOUND 800, .3
            SOUND 825, .3
        NEXT i%
    CASE 2                      ' European siren
        FOR i% = 1 TO 5
            SOUND 500, 10
            SOUND 450, 10
        NEXT i%
    CASE 3                      ' takeoff!
        FOR i% = 500 TO 2500
            SOUND i%, .03       ' uses value of i% for sound
        NEXT i%
END SELECT
```

2. Run the program. Your output screen will look like this:

```
SOUND-EFFECTS GENERATOR
Please choose the sound you'd like to hear
from the following menu:

1.  Ray gun
2.  European siren
3.  Takeoff!

Please enter 1, 2, or 3:
```

3. Enter the appropriate value to hear the sound of your choice.

Run the program several times to try out each of the sounds. Next, try experimenting with this program by changing some of the values in the various SOUND statements and running the program again. Notice that most of the duration values are small. Smaller duration values work well for sound effects, whereas you'll need to use larger values to create a melody.

Controlling the Count with STEP

As you've seen, when QBasic executes a FOR loop, it begins by assigning the *start* value to the variable following FOR and then loops through the block of statements until the value of the counter variable is greater than the value of *end*. Each time it encounters the NEXT statement, QBasic increments the value of the counter by 1.

You can tell QBasic to increment the counter by a value other than 1 if you use the STEP clause. Here's an example:

```
FOR i% = 5 TO 25 STEP 5
    PRINT "This message will print 5 times."
NEXT i%
```

When QBasic encounters the NEXT statement, it increments the counter by the number you've specified after STEP.

The number in the STEP clause doesn't have to be positive. If you place a negative number after STEP, QBasic *decrements* the value of the counter each time it encounters the NEXT statement.

The STEP value also affects the values you assign to *start* and *end*:

- If the STEP value is positive, the *start* value must be less than or equal to the *end* value.
- If the STEP value is negative, the *start* value must be *greater* than or equal to the *end* value.

Think about it: If you use a negative STEP value and *end* is greater than *start*, QBasic will assume that its work is done, jump over the block of statements without executing it, and continue executing the rest of the program.

Practice: Using STEP

1. Type in the FOR-6.BAS program (Figure 6-6).

```
' FOR-6.BAS
' This program demonstrates the use of STEP.

CLS

FOR i% = 15 TO 1 STEP -1
   COLOR i%
   PRINT "Falling!"
   SOUND (50 * i%), 1
NEXT i%
```

FIGURE 6-6.
FOR-6.BAS: Using STEP in a FOR statement.

2. Run the program. QBasic will print the word *Falling!* 15 times, each time in a different color, with accompanying sound.

Nesting FOR Loops

Nesting is the term that describes the practice of putting one loop within another. Here's an example of a nested FOR loop:

```
FOR i% = 1 TO 5
   FOR j% = 1 TO 5
      PRINT "You’ll see this message 25 times."
   NEXT j%
NEXT i%
```

Here's where indentation really helps you see what's going on. In a nested FOR loop, the inner FOR loop is actually a statement for the outer loop. That is, the outer FOR loop executes the inner loop as many times as you specify. In this example, the outer loop executes the inner loop a total of 5 times. Because the inner loop executes its statements 5 times, QBasic ends up executing the PRINT statement a total of 25 times.

Practice: Working with nested FOR loops

1. Type in the FOR-7.BAS program (Figure 6-7).

```
' FOR-7.BAS
' This program demonstrates nested FOR loops.

CLS

PRINT "The inner loop will print the colored numbers:"
PRINT

FOR i% = 1 TO 5
    COLOR 7            ' set outer loop color to white
    PRINT              ' print a blank line
    PRINT "Outer loop lap number"; i%
    FOR j% = 1 TO 10
        COLOR j%
        PRINT j%;
        SOUND (j% * 100), 1
    NEXT j%
NEXT i%
```

FIGURE 6-7.
FOR-7.BAS: Using nested FOR loops.

2. Run the program. When the program has finished running, your output screen will look like this:

```
The inner loop will print the colored numbers:

Outer loop lap number 1
 1  2  3  4  5  6  7  8  9  10
Outer loop lap number 2
 1  2  3  4  5  6  7  8  9  10
Outer loop lap number 3
 1  2  3  4  5  6  7  8  9  10
Outer loop lap number 4
 1  2  3  4  5  6  7  8  9  10
Outer loop lap number 5
 1  2  3  4  5  6  7  8  9  10
```

The SOUND statement within the inner loop created a sound after each of the inner loop's numbers was printed. If you want to, press a key to return to the View window and change some of the values. Try to anticipate what the changed program will do before you run it, and then run the program to see whether you were right.

THE WHILE STATEMENT

To design a loop that repeats as long as a specific condition is true, use the WHILE statement. The WHILE statement always ends with the WEND statement, as shown in the following syntax:

```
WHILE condition
   statements to be repeatedly executed
WEND
```

condition can be any conditional expression (such as *wage! = 11.50*, *year% <= 1959*, or *temperature% < 32*).

When it encounters a WHILE statement, QBasic evaluates *condition*:

- If *condition* is true, QBasic executes the block of statements between the WHILE and WEND statements and then checks *condition* again.
- If *condition* is false, QBasic jumps to the WEND statement and moves on to the rest of the program.

To use a WHILE loop successfully, you must make provisions *inside* the loop to alter the condition following the WHILE statement.

NOTE: Be sure that condition *eventually becomes false; if you don't, your loop will never stop, and the rest of your program will never run. (A loop that never stops is called an* infinite *or* endless *loop. To stop an endless loop in QBasic, hold down the Ctrl key and press Break.)*

Practice: Working with the WHILE loop

In the following program, all activity takes place within the loop. And because QBasic increments the value of *userNum%* each time it executes the loop, *userNum%* eventually becomes greater than 12. This creates a false condition and causes QBasic to stop executing the loop.

1. Type in the WHILE.BAS program (Figure 6-8).

```
' WHILE.BAS
' This program demonstrates the WHILE loop.

CLS

INPUT "Please enter a number from 1 through 10:  ", userNum%
PRINT

WHILE userNum% <= 12
    COLOR userNum%
    PRINT "The current value of userNum% is"; userNum%
    SOUND ((userNum% * 30) + 300), .5
    userNum% = userNum% + 1
    PRINT
WEND
```

FIGURE 6-8.
WHILE.BAS: Using a WHILE loop with a counter.

2. Run the program. Your output screen should look something like this:

```
Please enter a number from 1 through 10:  9

The current value of userNum% is 9

The current value of userNum% is 10

The current value of userNum% is 11

The current value of userNum% is 12
```

Nesting WHILE Loops

Just as you can nest FOR statements, you can place one WHILE statement inside another. Each WHILE statement has its own *condition* and closing WEND statement; the two conditions need not be related to each other.

The syntax for a nested WHILE loop is as follows:

```
WHILE condition
    WHILE condition
        statement for inner WHILE loop
    WEND
WEND
```

The inner WHILE loop serves as the statement of the outer WHILE loop.

NOTE: To prevent infinite loops, remember that both the inner condition and the outer condition must eventually evaluate as false.

THE DO STATEMENT

You use the DO statement together with the LOOP statement to develop two kinds of loops:

- A loop that repeats *as long as* a certain condition is true (DO WHILE)
- A loop that repeats *until* a certain condition *becomes* true (DO UNTIL)

DO WHILE: Looping as Long as a Condition Is True

To create a loop that repeats as long as a condition is true, use a DO loop with the following syntax:

```
DO WHILE condition
    block of statements to be executed
LOOP
```

A DO loop of this type works just as a WHILE statement does. QBasic examines *condition*:

- If *condition* is true, QBasic executes the statements and continues to do so as long as *condition* is true.
- If *condition* is false, QBasic jumps past the LOOP statement to execute the rest of the program.

The Condition-Statement Connection

As you learned in Chapter 2, QBasic executes instructions in the order in which they appear in your programs. The importance of order is particularly apparent within the DO statement, which requires you to place a condition in one of two places, based on the result you want from your program:

- If you want the statements to execute based solely on the value of *condition*, place *condition* on the line with the DO statement:

  ```
  DO {WHILE¦UNTIL} condition
     block of statements to be executed
  LOOP
  ```

  In this form, the inner statements are executed only if *condition* is true.

- If you want the statements to execute at least once, no matter what the value of *condition*, place *condition* on the line with the LOOP statement:

  ```
  DO
     block of statements to be executed
  LOOP {WHILE¦UNTIL} condition
  ```

  By the time *condition* is evaluated, QBasic has already executed the statements one time.

Practice: Using a DO WHILE loop with the condition at the top

1. Type in the DO-1.BAS program (Figure 6-9).

```
' DO-1.BAS
' This program demonstrates a DO WHILE loop.  Notice
'   that the condition is at the top of the loop.

CLS

INPUT "Please enter a number from 10 through 15:  ", userNum%
PRINT

DO WHILE userNum% >= 10
    COLOR userNum%
    PRINT "The value of userNum% is"; userNum%
    userNum% = userNum% - 1   ' decrement userNum% by 1
LOOP
```

FIGURE 6-9.
DO-1.BAS: Using a DO WHILE loop.

2. Run the program. Your output screen should look something like this:

```
Please enter a number from 10 through 15:  13

The value of userNum% is 13
The value of userNum% is 12
The value of userNum% is 11
The value of userNum% is 10
```

3. Run the program again, but this time enter a value of 9 or less. Your output screen will look something like this:

```
Please enter a number from 10 through 15:  6
```

   QBasic perceived the condition was false and therefore skipped the statements within the loop.

Practice: Using a DO WHILE loop with the condition at the bottom

Program DO-2.BAS is identical to DO-1.BAS, but the condition has been moved to the bottom of the loop.

1. Modify the DO-1.BAS program to match the DO-2.BAS program (Figure 6-10).

```
' DO-2.BAS
' This program demonstrates a DO WHILE loop.  Notice
'   that the condition is at the bottom of the loop.

CLS

INPUT "Please enter a number from 10 through 15:  ", userNum%
PRINT

DO
    COLOR userNum%
    PRINT "The value of userNum% is"; userNum%
    userNum% = userNum% - 1   ' decrement userNum% by 1
LOOP WHILE userNum% >= 10
```

FIGURE 6-10.
DO-2.BAS: A DO WHILE loop with the condition at the bottom.

2. Run the program. Your output screen should look something like this:

```
Please enter a number from 10 through 15:  13

The value of userNum% is 13
The value of userNum% is 12
The value of userNum% is 11
The value of userNum% is 10
```

Looks just like the output of DO-1.BAS, doesn't it? But wait.

3. Run the program again, but this time enter a value of 9 or less. Your output screen will look something like this:

```
Please enter a number from 10 through 15:  6

The value of userNum% is 6
```

Here's where you can see the difference. Even though the value of *userNum%* caused the condition in the LOOP statement to be false, QBasic didn't encounter that condition until *after* it had executed the body of the DO loop once.

DO UNTIL: Looping Until a Condition Becomes True

To create a loop that executes until a condition *becomes* true, use a DO loop with the following syntax:

```
DO UNTIL condition
    block of statements to be executed
LOOP
```

QBasic examines *condition*:

- If *condition* is true, QBasic jumps past the LOOP statement to execute the rest of the program.
- If *condition* is false, QBasic executes the statements and continues to do so until *condition* is true.

Practice: Using a DO UNTIL loop with the condition at the top

1. Type in the DO-3.BAS program (Figure 6-11).

```
' DO-3.BAS
' This program demonstrates a DO UNTIL loop.  Notice
'   that the condition is at the top of the loop.

CLS

INPUT "Please enter a number from 1 through 10:  ", userNum%
PRINT

DO UNTIL userNum% > 10
    COLOR userNum%
    PRINT "The value of userNum% is"; userNum%
    userNum% = userNum% + 1   ' increment userNum% by 1
LOOP
```

FIGURE 6-11.
DO-3.BAS: Using a DO UNTIL loop.

2. Run the program. Your output screen should look something like this:

```
Please enter a number from 1 through 10:  7

The value of userNum% is 7
The value of userNum% is 8
The value of userNum% is 9
The value of userNum% is 10
```

   Because the condition *userNum% > 10* was false when QBasic first encountered the loop, QBasic continued to execute the loop until the condition became true.

3. Run the program again, this time entering a value of 11 or more. Your output screen will look something like this:

```
Please enter a number from 1 through 10:  13
```

   This time, because the condition was true when QBasic encountered the loop, QBasic jumped over the loop and executed the rest of the program.

Practice: Using a DO UNTIL loop with the condition at the bottom

1. Modify the DO-3.BAS program to match the DO-4.BAS program (Figure 6-12).
2. Run the program. Your output screen should look something like this:

```
Please enter a number from 1 through 10:  7

The value of userNum% is 7
The value of userNum% is 8
The value of userNum% is 9
The value of userNum% is 10
```

   The output of this program is the same as that for the previous program.

```
' DO-4.BAS
' This program demonstrates a DO UNTIL loop.  Notice
'   that the condition is at the bottom of the loop.

CLS

INPUT "Please enter a number from 1 through 10:  ", userNum%
PRINT

DO
    COLOR userNum%
    PRINT "The value of userNum% is"; userNum%
    userNum% = userNum% + 1   ' increment userNum% by 1
LOOP UNTIL userNum% > 10
```

FIGURE 6-12.
DO-4.BAS: A DO UNTIL loop with the condition at the bottom.

3. Run the program again, but this time enter a value of 11 or more. Your output screen will look something like this:

```
Please enter a number from 1 through 10:  13

The value of userNum% is 13
```

This time, because the condition was at the bottom of the loop, QBasic executed the contents of the loop once because it did not know what the condition was until it got to the bottom of the loop.

Nesting DO Loops

You can nest DO loops the same way you nest FOR and WHILE loops.

The loops are totally unrelated in terms of how you set them up. In other words, if you have two nested DO loops and you put the condition at the top of the outer loop, you needn't put the condition at the top of the inner loop. Of course, as with a single DO loop, you must ensure that both the outer loop and the nested loop will eventually terminate.

NESTING DIFFERENT KINDS OF LOOPS

So far we've discussed nesting loops of the same kind—a FOR loop inside another FOR loop, a WHILE loop inside another WHILE loop, and so on.

You can mix the kinds of loops, too, especially when that will make your program more efficient. The following program nests a FOR loop inside a DO loop. The DO loop allows you to execute the FOR loop as many times as you want to without having to start the program each time.

Practice: Nesting different kinds of loops

1. Type in the NESTING.BAS program (Figure 6-13).

```
' NESTING.BAS
' This program demonstrates nesting different kinds of loops.

CLS

DO
    PRINT "How many tones do you want to hear?"
    INPUT "(Enter 0 to end)  ", numSounds%
    PRINT

    FOR i% = 1 TO numSounds%
        SOUND (i% * 100), 1
    NEXT i%

LOOP UNTIL numSounds% = 0
```

FIGURE 6-13.
NESTING.BAS: Nesting different kinds of loops.

2. Run the program. Your output screen should look something like this:

```
How many tones do you want to hear?
(Enter 0 to end)  5

How many tones do you want to hear?
(Enter 0 to end)  20

How many tones do you want to hear?
(Enter 0 to end)  0
```

PRACTICAL USES FOR QBASIC LOOPS

Now that you've been introduced to loops, it's time to see what they can really do for you. The following examples present some practical and fun applications for loops.

Using a Loop to Collect Information

Let's say that you want to automate your monthly budget planning. The philosophy behind budgeting is straightforward: You subtract monthly expenses from monthly income.

Using the tools you've learned about so far, you could write a QBasic program to help you do this. You might use several INPUT statements: one to get the starting amount, and then one for each expenditure. After some calculations, you could have QBasic print out how much money you'd have left. The program shown in Figure 6-14 is an example of such a program.

Which Loop Should You Choose?

Keep the following points in mind when deciding which type of loop to use in a program:

- Use a FOR loop when you want to execute a block of statements a specific number of times.
- Use a WHILE or DO loop to execute statements based on the value of a condition.
- You can make a DO loop do anything a WHILE loop can. Get into the habit of using the more versatile DO loop whenever possible. (QBasic supports the WHILE loop so that you can run programs that were written for earlier versions of BASIC.)

```
' This program helps you figure out how much money is
'   left over after you've paid your monthly bills.

CLS

COLOR 2                 ' green foreground
PRINT "Budget Calculator"
COLOR 7                 ' restore default white color
PRINT
INPUT "Enter your total income for this month:  $", total!
PRINT

INPUT "Enter expense # 1:  $", expense1!
total! = total! - expense1!
INPUT "Enter expense # 2:  $", expense2!
total! = total! - expense2!
INPUT "Enter expense # 3:  $", expense3!
total! = total! - expense3!
INPUT "Enter expense # 4:  $", expense4!
total! = total! - expense4!

PRINT
PRINT "You have $"; total!; "left over this month."
```

FIGURE 6-14.
A sample budgeting program.

This program would work fine, but it does have limitations:

- Because your number of expenditures might change from month to month, you'd have to alter the program each time you ran it.
- Each expenditure you enter must be immediately subtracted from the running total. This isn't a problem, but it is a lot of repetitive calculation. Consider how long this program would be if you had 20 or more expenses each month!

Programs with this degree of repetition are ideal candidates for a loop. The BUDGET.BAS program in Figure 6-15 does the same job as the program in Figure 6-14 but offers some advantages:

- It's a shorter program.
- It accommodates a changing number of expenditures by asking you for the total number of bills you plan to enter.
- It uses a loop both to ask you for the bill amount and to subtract the amount from the running total.

```
' BUDGET.BAS
' This program helps you figure out how much money is
'   left over after you've paid your monthly bills.

CLS

COLOR 2                 ' green foreground
PRINT "Budget Calculator"
COLOR 7                 ' restore default white foreground
PRINT
INPUT "Enter your total income for this month:  $", total!
PRINT
INPUT "How many bills will you have this month?  ", bills%

FOR i% = 1 TO bills%
    PRINT "Enter expense #"; i%;
    INPUT ":  $", thisExpense!
    total! = total! - thisExpense!
NEXT i%

PRINT
PRINT "You have $"; total!; "left over this month."
```

FIGURE 6-15.
BUDGET.BAS: A better budgeting program.

Using Loops with Random Numbers

Dice. Roulette. Keno. Bingo. Lottery. All of these games make use of *random numbers*—numbers that occur in no predictable order. We'll discuss three of the tools that QBasic provides for creating random numbers: RND, RANDOMIZE TIMER, and INT.

Creating random numbers

The RND function tells QBasic to return a random number to your program. The number will be a single-precision floating-point number between 0 and 1. RND is a function, so you must use it within a QBasic statement. Here's the syntax for the simplest form of the RND function:

```
RND
```

By itself, RND returns the same series of numbers whenever you run your program. To create a *different* series of numbers each time, use RANDOMIZE TIMER as one of the first statements in your program. The syntax for the RANDOMIZE TIMER statement is as simple as that for the RND function:

```
RANDOMIZE TIMER
```

Practice: Creating random numbers

The RND-1.BAS program (Figure 6-16) uses a FOR loop to create six random numbers. (To create more random numbers, simply change the *6* in the FOR statement to a higher value.)

1. Type in the RND-1.BAS program.

```
' RND-1.BAS
' This program generates random numbers between 0 and 1.

CLS

RANDOMIZE TIMER

FOR i% = 1 TO 6
   PRINT RND
NEXT i%
```

FIGURE 6-16.
RND-1.BAS: Creating random numbers.

2. Run the program. Your output screen will look something like this:

```
.2494013
.3081127
3.670245E-02
.574261
.4791026
.1495265
```

The numbers you get will undoubtedly be different from these—remember, these are just random numbers.

Customizing the results of RND

Because numbers between 0 and 1 might not always meet your needs, you can use QBasic's mathematical operators to change the size of RND's results.

Practice: Creating larger random numbers

The RND-2.BAS program (Figure 6-17) is identical to RND-1.BAS, but it multiplies the result of RND by 100, creating values that range between 0 and 100.

1. Modify the RND-1.BAS program to match the RND-2.BAS program.

```
' RND-2.BAS
' This program generates random numbers between 0 and 100.

CLS

RANDOMIZE TIMER

FOR i% = 1 TO 6
   PRINT RND * 100
NEXT i%
```

FIGURE 6-17.
RND-2.BAS: Creating larger random numbers.

2. Run the program. Your output screen should look something like this:

```
85.67621
42.35302
87.795
48.80183
42.53306
24.83424
```

Very Small and Very Large Numbers in QBasic

The values produced by QBasic's RND function are always greater than 0 and less than 1. Sometimes the values are very small.

When QBasic tries to print a value that's too small or too large to display without using a lot of digits, it switches to QBasic's version of exponential notation. To put it simply, when you see an *E* or a *D* in a number, the decimal point was shifted from its actual location. The number after the *E* or *D* indicates in which direction and by how many places the decimal point was moved. If that number is positive, the decimal point actually resides to the *right* of its displayed location; if the number is negative, the decimal point belongs to the *left* of its displayed location.

For example, in the sample output from RND-1.BAS you can see the number *3.670245E-02*. The *E-02* means that the decimal point actually belongs two places to the left. In other words, you'd normally write the number as 0.03670245.

So why not simply display the number as it usually appears? Often the number won't fit neatly on the screen. Take the number *4.136111E28*. If printed in its entirety, it would look like this:

```
41361110000000000000000000000
```

For those of you who are familiar with standard scientific notation, here is the same number in that format:

4.136111×10^{28}

Creating random integers

If you'd like to create random numbers that are integers (numbers without decimal points), use the INT function along with the RND function. INT discards the fractional portion of a floating-point number, leaving only the integer portion. It takes the following form:

```
INT(number)
```

number is the floating-point number you want to round. As with all functions that return integers, the integer result of INT must be assigned either to a statement that accepts integer values or to an integer variable.

The following practice session demonstrates how to use the INT function with RND to create random integer numbers.

Practice: Creating a guess-a-number game

Random numbers are ideal for guessing games. The following program uses a DO loop to demonstrate this:

1. Type in the GUESS.BAS program (Figure 6-18).
2. Run the program. Your output screen should look something like this:

```
Guess-a-number game

I'm thinking of a number from 1 through 100.
Can you guess what it is?

What is your guess?  50
Try a bigger number!

What is your guess?  99
Try a smaller number!

What is your guess?  65
Try a bigger number!

What is your guess?  71
That's right!!!

You guessed the number in 4 tries!
```

```
' GUESS.BAS
' This program is a guess-a-number game.  The program generates a
'   random number and asks the user to guess what it is.  After the
'   user has guessed the number, the program displays the number
'   of guesses made.

CLS

PRINT "Guess-a-number game"
PRINT
PRINT "I'm thinking of a number from 1 through 100."
PRINT "Can you guess what it is?"
PRINT

RANDOMIZE TIMER

randNum% = INT(RND * 100) + 1            ' generate random number
numGuesses% = 0                          ' start with a clean slate

DO
    INPUT "What is your guess?  ", guess%
    IF guess% = randNum% THEN PRINT "That’s right!!!"
    IF guess% < randNum% THEN PRINT "Try a bigger number!"
    IF guess% > randNum% THEN PRINT "Try a smaller number!"

    numGuesses% = numGuesses% + 1        ' chalk up one guess
    PRINT                                ' print a blank line

LOOP UNTIL guess% = randNum%

PRINT "You guessed the number in"; numGuesses%; "tries!"
```

FIGURE 6-18.
GUESS.BAS: A guessing game.

Practice: Writing a computer simulation

Let's say that you want to write a program that calculates how many times, out of 100 rolls of 2 dice, the number 7 comes up. The DICE.BAS program (Figure 6-19) does just that, and the RND function comes in handy again.

1. Type in the DICE.BAS program.

```
' DICE.BAS
' This program asks the user to enter how many times QBasic
'   should "roll" two dice, and then calculates how many
'   times the number 7 comes up.

CLS

RANDOMIZE TIMER

COLOR 4                 ' red color for title
PRINT "Dice-Simulation Program"
COLOR 7                 ' restore default white color
PRINT
numSeven% = 0           ' start with 7 counter equal to 0

INPUT "How many times should I roll the dice?  ", rolls%
PRINT
PRINT "Working..."  ' ensure the user knows nothing is wrong

FOR i% = 1 TO rolls%
    die1% = INT(RND * 6) + 1    ' "roll" the first die
    die2% = INT(RND * 6) + 1    ' "roll" the second die
    IF die1% + die2% = 7 THEN numSeven% = numSeven% + 1
NEXT i%

PRINT
PRINT "Out of"; rolls%; "rolls, the number 7 came"
PRINT "up"; numSeven%; "times."
```

FIGURE 6-19.

DICE.BAS: A dice-simulation program.

2. Run the program. You should see output similar to this:

```
Dice-Simulation Program

How many times should I roll the dice?  50

Working...

Out of 50 rolls, the number 7 came
up 10 times.
```

SUMMARY

Loops let you create powerful programs that repeat a task a specific number of times or until a condition is met. In this chapter you've learned about FOR, WHILE, and DO loops. Combine this knowledge with the COLOR and SOUND statements, the RND function, and the skills you've picked up in previous chapters, and you're ready to take on some impressive programming projects. The following chapters will get you started.

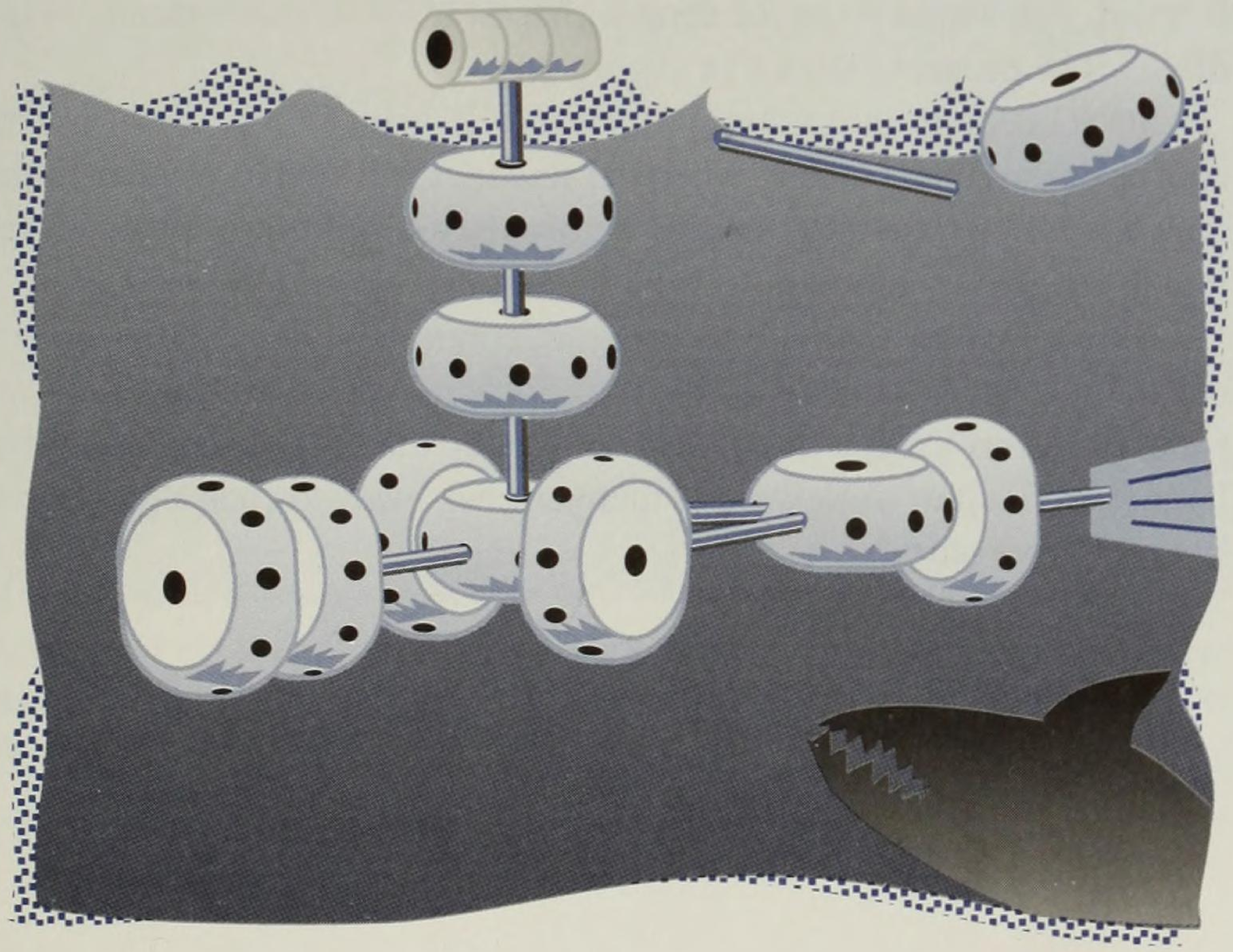
QUESTIONS AND EXERCISES

1. What is the purpose of the counter variable in a FOR loop?
2. What types of values can be assigned to the *start* and *end* elements of a FOR loop?
3. What number would you specify as an argument to the COLOR statement if you wanted to set the foreground color to red?
4. What does the SOUND statement do?
5. What is a nested loop?
6. What is the effect of putting the condition in a DO loop at the top of the loop? What is the effect of putting the condition at the bottom of the loop?
7. Under what circumstances might you use a FOR loop? A WHILE or DO loop?

8. Write a program that keeps track of the amount of money spent on gasoline in a week. Use a FOR loop to collect the dollars and cents spent, and keep a running total in a single-precision floating-point variable.
9. Write a program that prompts the user for valid frequency and duration values and plays them back with the SOUND statement. Use a DO UNTIL loop to collect information until the user types in *–999* as the frequency.
10. Write a program that rolls one simulated die 10 times. Print the value of the die after each roll, and display the message *Nice Roll!* in green if the die shows 6.

CHAPTER 7

Creating Your Own Subprograms and Functions



Up to now, you've written relatively short programs—none has been longer than 30 lines. But now that you've learned the basics of QBasic, you're ready to write longer programs. In this chapter, you'll learn some techniques that allow you to write longer programs with a minimum of extra time and effort.

You'll learn about two program structures: *subprograms* and *functions*. (You'll see how these functions differ from the built-in functions you've been using.) You use these structures to separate a program into easy-to-use units called *procedures*. We'll show you how to declare subprograms and functions and how to use them in your programs. You'll also learn how the QBasic environment is designed for modular programming. By the end of this chapter, you'll have all the tools you need to write compact, well-organized programs.

WHY PROCEDURES?

Suppose you want to write a program that prints the lyrics of the traditional American song ''Clementine.'' Using the skills you've already learned, you would write the song with a lot of PRINT statements, as in the SONG.BAS program shown in Figure 7-1.

SONG.BAS is quite straightforward: It prints each verse and each chorus of the song, pausing when the screen is full so that you have time to read the lyrics (and sing along!) before they scroll out of sight. If you'd like to see this in action now, type in the program and give it a try.

Notice that the SONG.BAS program contains a chorus that is repeated over and over without modification. Such repetitive text not only takes time and effort to type in but also clutters up the program listing, making it more difficult to work with. Is there an easier way to code a program that has repetitive parts?

```
' SONG.BAS
' This program displays the lyrics of the folk song "Clementine."

CLS
PRINT "------------------------- Clementine -------------------------"
PRINT

PRINT "In a cavern, in a canyon,"              ' first verse
PRINT "Excavating for a mine,"
PRINT "Lived a miner, forty-niner,"
PRINT "And his daughter, Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

PRINT "Light she was and like a fairy,"        ' second verse
PRINT "And her shoes were number nine;"
PRINT "Herring boxes without topses,"
PRINT "Sandals were for Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Enter for more...", dummy$        ' pause
PRINT
```

FIGURE 7-1. *(continued)*

SONG.BAS: A program that prints the lyrics to the song "Clementine" using a series of PRINT statements.

FIGURE 7-1. *continued*

```
PRINT "Drove she ducklings to the water,"      ' third verse
PRINT "Ev'ry morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

PRINT "Ruby lips above the water,"             ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Enter for more...", dummy$        ' pause
PRINT

PRINT "Then the miner, forty-niner,"           ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT
```

The Procedure Advantage

The answer is *Yes!* QBasic provides a programming structure called a *procedure* that lets you type a block of statements, assign a name to it, and then call it by name whenever you want your program to execute it. Procedures provide the following advantages:

- **Procedures eliminate repeated lines.** You can define a procedure once and have your program execute it any number of times.
- **Procedures make programs easier to read.** A program divided into a collection of smaller parts is easier to take apart and understand.
- **Procedures simplify program development.** Programs separated into logical units are easier to design, write, and debug. Plus, if you're writing a program with a friend, you can exchange procedures instead of entire programs.
- **Procedures can be reused in other programs.** You can incorporate your general-purpose procedures into other programming projects.
- **Procedures extend the QBasic language.** Procedures can often perform tasks that can't be accomplished directly by built-in QBasic statements and functions.

In this chapter you'll learn to create and use two QBasic procedures: *subprograms* and *functions.* Subprograms let you subdivide your program into smaller units that can be called one or more times. Functions return values that your program can use.

CREATING SUBPROGRAMS

Subprograms make your code easy to read and reduce repetition. A subprogram is a block of code between SUB and END SUB statements. You can have your program call a subprogram as often as you like. When a subprogram ends, control returns to the statement that follows the subprogram call in the main program. In QBasic, you create and store subprograms separately from main programs.

Syntax of a Subprogram

The syntax of a subprogram is as follows:

```
SUB SubprogramName (parameterList)
    local variable and constant declarations
    subprogram statements
END SUB
```

- SUB is the QBasic statement that marks the beginning of the subprogram definition.
- *SubprogramName* is the name of the subprogram. It can be as many as 40 characters long and is the name the main program uses to call the subprogram. It can't be a QBasic keyword or be the same as any variable name or procedure name in your program.
- (*parameterList*) is an optional list of variables. (See "Passing Arguments to a Subprogram," later in this chapter.) If you use *parameterList*, you must enclose it in parentheses.
- *local variable and constant declarations* is an optional list of variables and constants declared and used within—and only within—the subprogram. They have no effect on variables or constants with the same name elsewhere in the program.
- *subprogram statements* is the working part of the subprogram. You can use almost any BASIC statements in a subprogram.
- END SUB is the QBasic statement that marks the end of the subprogram definition.

To see how these elements work together, examine the following subprogram named *Chorus*. Every time *Chorus* is executed, it prints a blank line, the four-line chorus to "Clementine," and another blank line.

```
SUB Chorus

' The Chorus subprogram prints the chorus of the song "Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"        ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

END SUB
```

Creating a Subprogram

In the QBasic environment, menus help you create and work with subprograms quickly and easily. QBasic provides two menu commands for working with subprograms:

- The New SUB command (on the Edit menu)
- The SUBs command (on the View menu)

The New SUB command

The New SUB command is the QBasic menu command you use to add a subprogram to your program. To create a subprogram, you perform the following steps:

1. Select the New SUB command. The New SUB dialog box appears.
2. Enter the name of the subprogram (if it isn't already there).
3. Type in the parameters and body of the subprogram.

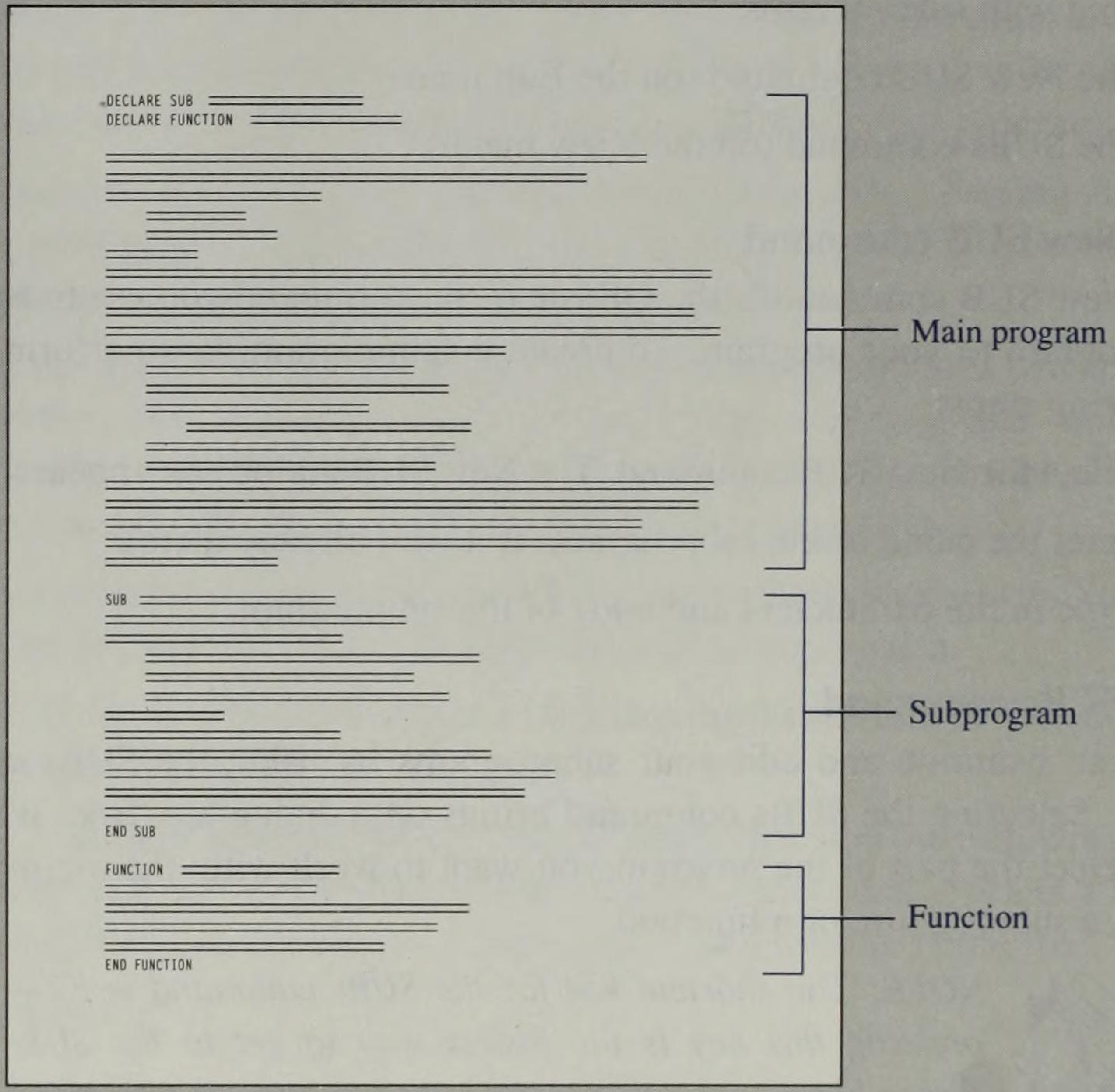
The SUBs command

You can examine and edit your subprograms by using the SUBs command. Selecting the SUBs command brings up a dialog box from which you select the part of the program you want to work with: the main program, a subprogram, or a function.

NOTE: The shortcut key for the SUBs command is F2—pressing this key is the fastest way to get to the SUBs dialog box.

Dividing Up a Program

This chapter describes how to organize a program into three self-contained blocks of code: the *main program,* the *subprograms,* and the *functions.* You can have any number of subprograms and functions (collectively called *procedures*) in your program or none at all. Every program, however, must have a main program section that controls the general flow of the program and calls the subprograms and functions.


DECLARE SUB
DECLARE FUNCTION
Main program
SUB
Subprogram
END SUB
FUNCTION
Function
END FUNCTION

Although it might seem like a little extra work at first, learning to divide your programs into these organizational units will pay off when you start writing longer programs. Remember: No matter how many subprograms and functions a program contains, it is still a single QBasic program.

The DECLARE Statement

One of the advantages of QBasic is that it helps you program properly. For example, based on the information you provide when you create a subprogram, QBasic adds a DECLARE statement to the top of your main program when you save the program on disk. A DECLARE statement *declares* the existence of a subprogram to the main program. Each subprogram must have a corresponding DECLARE statement at the top of the main program.

The syntax for the DECLARE statement is as follows:

```
DECLARE SUB SubprogramName (parameters)
```

SubprogramName is the name of the subprogram, and *parameters* is a list of variables received by the subprogram.

NOTE: Although QBasic generates the DECLARE statement on its own, it's up to you to maintain the parameter list. If you change the parameter list in your subprogram, you must change the parameter list in the DECLARE statement to match. See the section "Passing Arguments to a Subprogram," later in this chapter, for more information.

By convention, DECLARE statements appear after the introductory comments in a program and before constant and variable declarations. In addition, QBasic requires that the DECLARE statements precede any executable statements.

Calling a Subprogram

After you've created and declared a subprogram, you can *call* (execute) it. To call a subprogram, simply specify the subprogram name—along with any arguments the subprogram requires—in the body of the main program or another procedure.

The SONG2.BAS program (Figure 7-2) demonstrates how to declare and call the *Chorus* subprogram. SONG2.BAS is nine lines shorter and is easier to follow than SONG.BAS. Note how the subprogram *Chorus* is set off from the rest of the program starting at line 53 by SUB and END SUB statements. The main program contains five calls to the *Chorus* subprogram. Note also that the *Chorus* subprogram name begins with an uppercase letter—we'll use this convention throughout this book to distinguish procedure names from variable names and from other QBasic statements and functions.

NOTE: Figure 7-2 contains a listing of the SONG2.BAS program as it would be printed on your printer or listed in a book—not as it appears in the QBasic environment. Remember that inside QBasic all subprograms are separated from the main program and are accessible through the SUBs command on the View menu. To type this program in from scratch, follow the instructions in the next practice session.

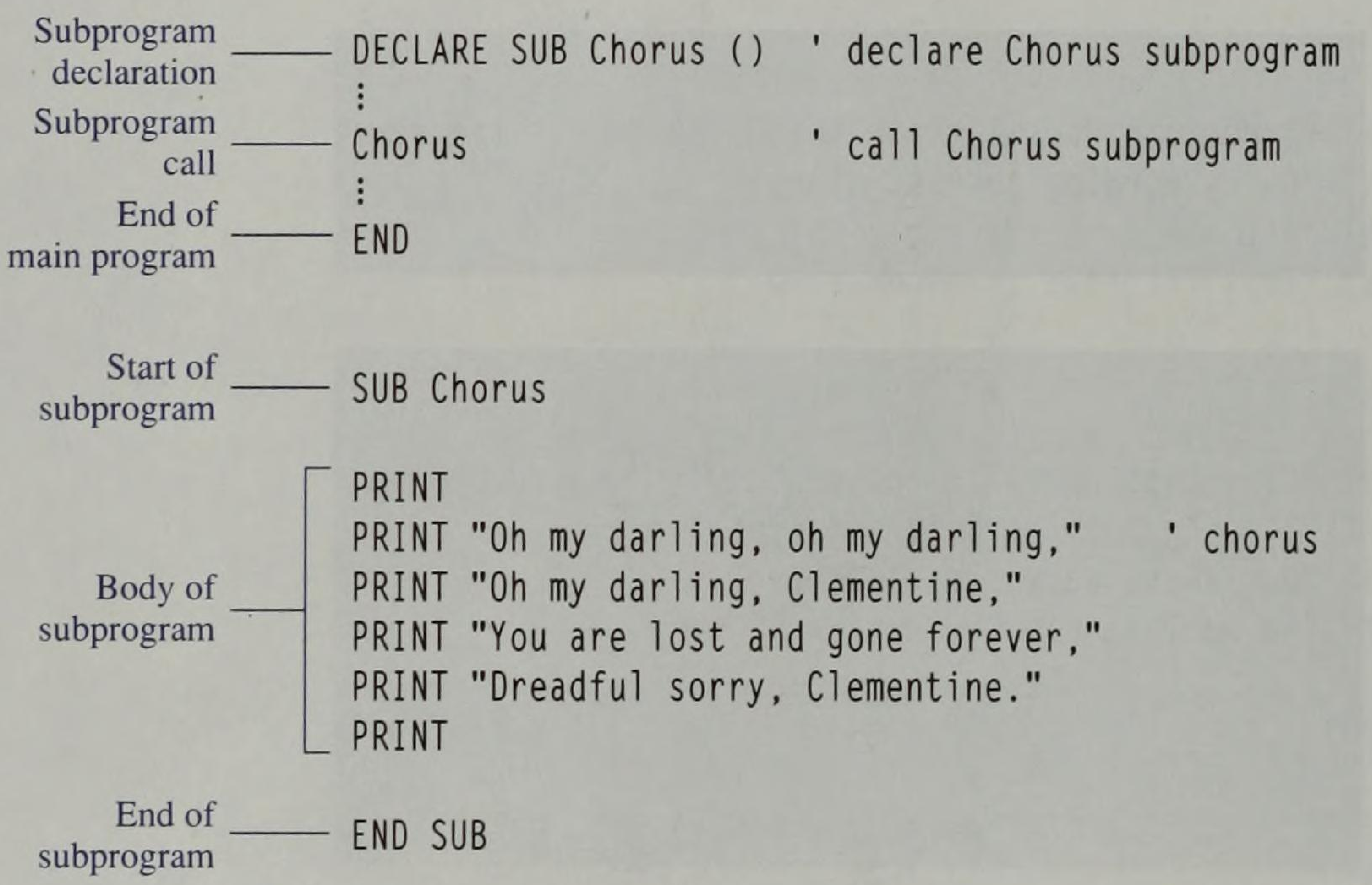
```
1   ' SONG2.BAS
2   ' This program displays the lyrics of the folk song "Clementine."
3
4   DECLARE SUB Chorus ()   ' declare Chorus subprogram before use
5
6   CLS
7   PRINT "------------------------ Clementine ------------------------"
8   PRINT
9
10  PRINT "In a cavern, in a canyon,"         ' first verse
11  PRINT "Excavating for a mine,"
12  PRINT "Lived a miner, forty-niner,"
13  PRINT "And his daughter, Clementine."
14
15  Chorus                                    ' call Chorus subprogram
16
17  PRINT "Light she was and like a fairy,"   ' second verse
18  PRINT "And her shoes were number nine;"
19  PRINT "Herring boxes without topses,"
20  PRINT "Sandals were for Clementine."
21
22  Chorus                                    ' call Chorus subprogram
23
```

FIGURE 7-2. *(continued)*
SONG2.BAS: A program that prints the lyrics to the song "Clementine" using a subprogram named Chorus.

FIGURE 7-2. *continued*

```
INPUT "Press Enter for more...", dummy$   ' pause
PRINT

PRINT "Drove she ducklings to the water," ' third verse
PRINT "Ev'ry morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."

Chorus                                    ' call Chorus subprogram

PRINT "Ruby lips above the water,"        ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."

Chorus                                    ' call Chorus subprogram

INPUT "Press Enter for more...", dummy$   ' pause
PRINT

PRINT "Then the miner, forty-niner,"      ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."

Chorus                                    ' call Chorus subprogram

END

SUB Chorus

' The Chorus subprogram prints the chorus of the song "Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"     ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

END SUB
```

Figure 7-3 provides a detailed look at the *Chorus* subprogram definition and call.



```
DECLARE SUB Chorus ()   ' declare Chorus subprogram
:
Chorus                  ' call Chorus subprogram
:
END

SUB Chorus

PRINT
PRINT "Oh my darling, oh my darling,"     ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

END SUB
```

FIGURE 7-3.
The components of a typical subprogram definition and call.

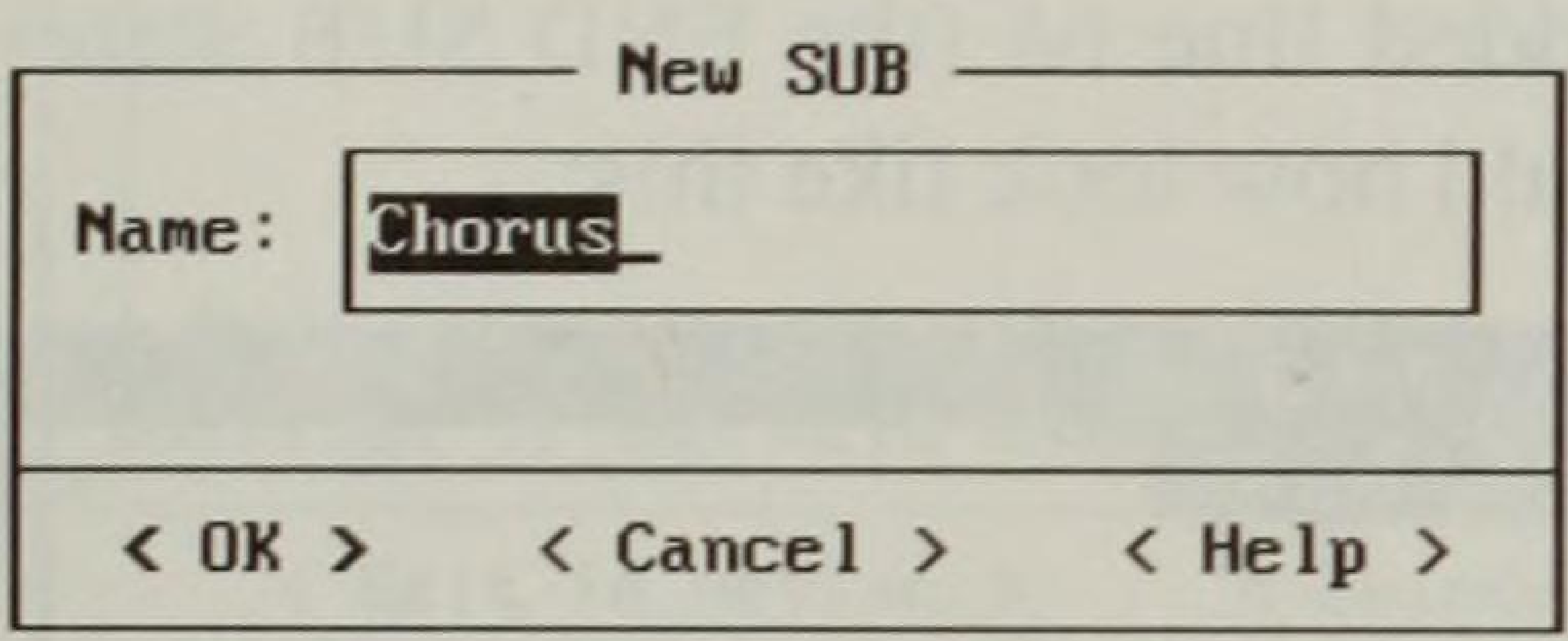
Practice: Entering the SONG2.BAS program

The following exercise walks you through the steps necessary to enter the SONG2.BAS program (Figure 7-2). You should adapt and follow these steps whenever you add a new subprogram to your program.

The END Statement

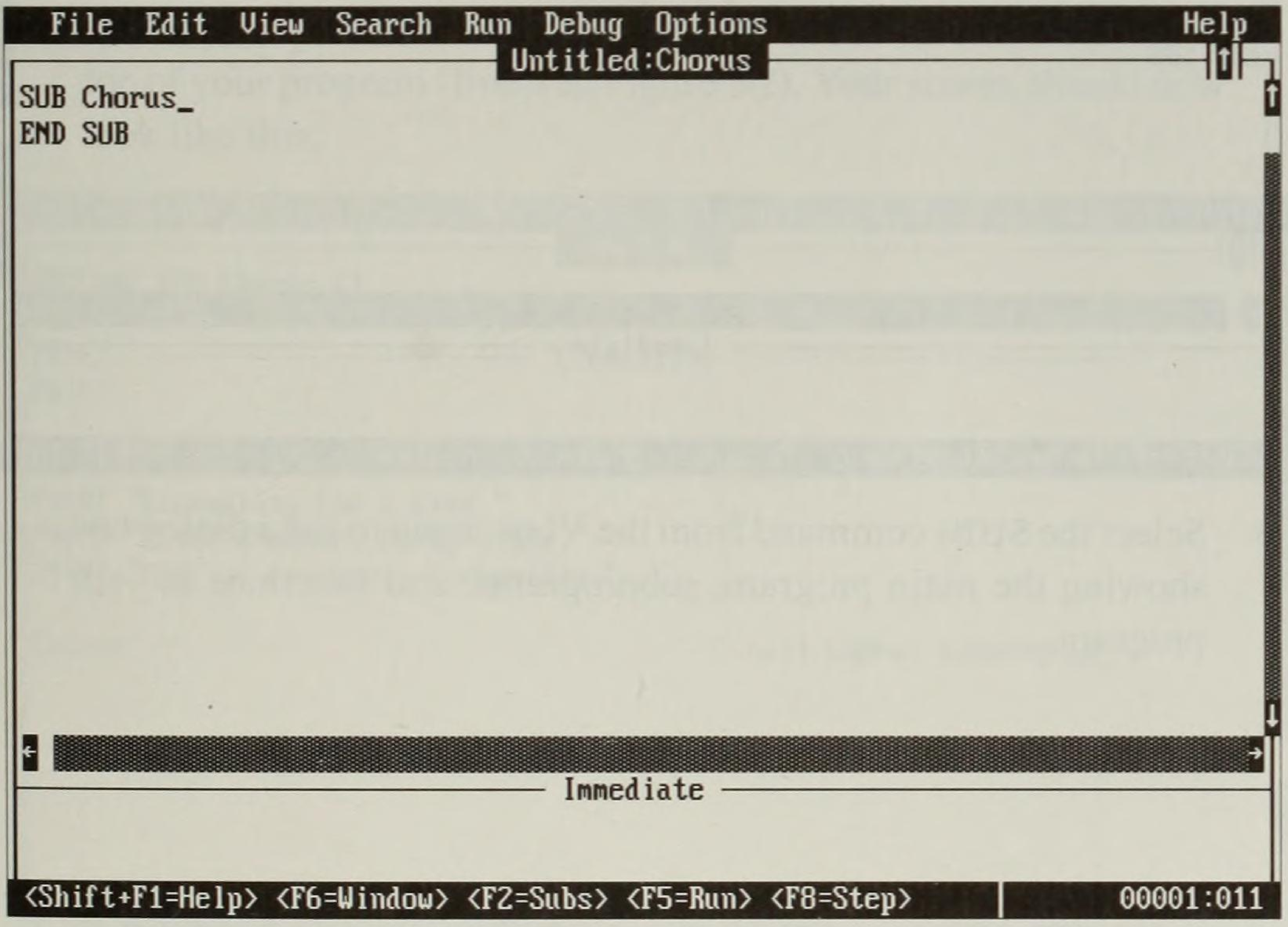
Throughout this book you'll see an END statement at the end of the main program section when a program contains subprograms or functions. END is an optional instruction that tells QBasic that the last instruction in the program has been reached and that execution should stop. Although QBasic doesn't require the END statement, END is a useful visual clue indicating exactly where the main program section ends.

1. Choose New from the File menu to start a new program, and type in lines 6 through 15 of Figure 7-2.
2. Place the cursor on the call to the *Chorus* subprogram (the word *Chorus*), and select New SUB from the Edit menu. The following dialog box will appear:


New SUB
< OK >
< Cancel >
< Help >

Name: Chorus_

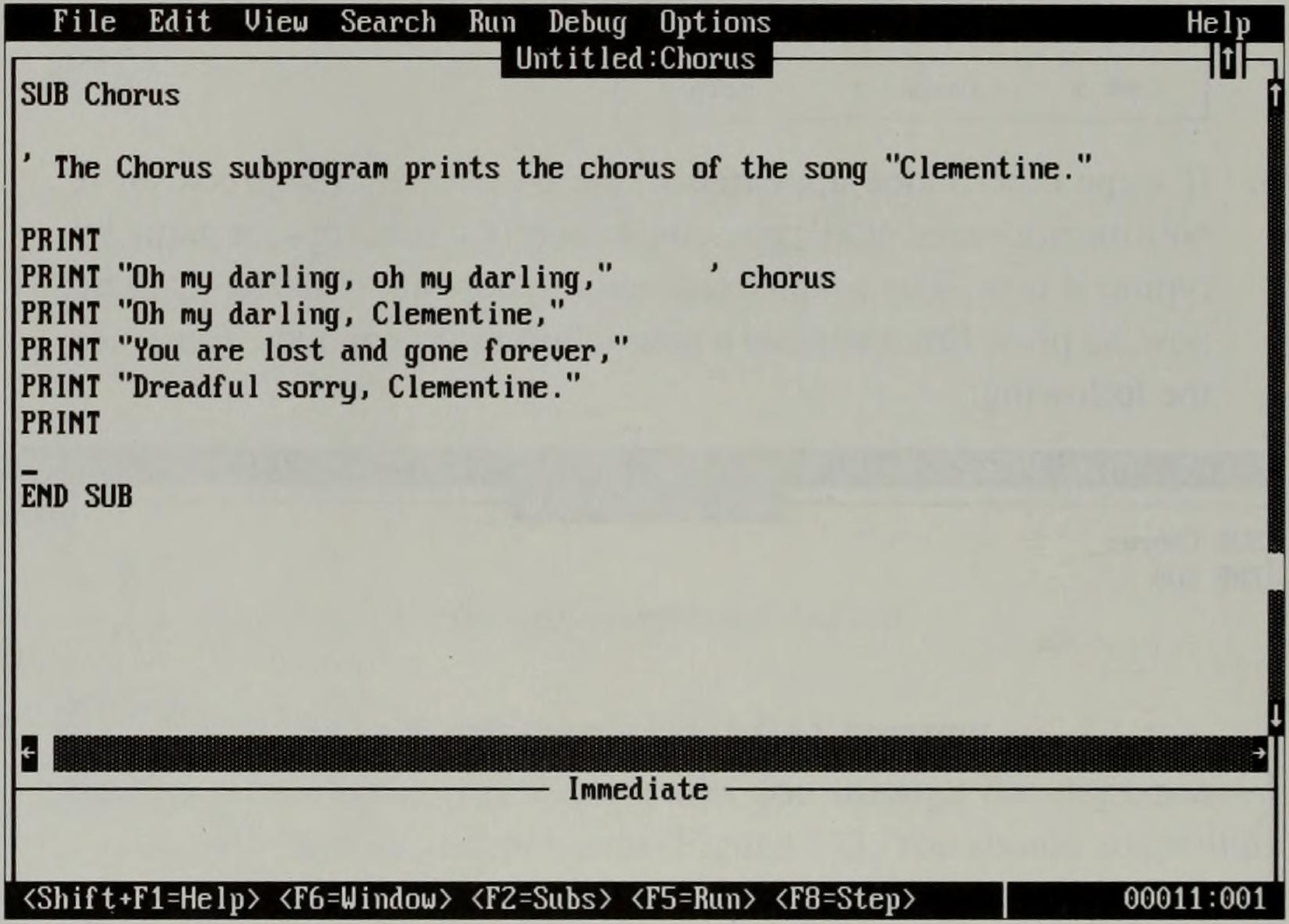
3. If some other name appeared in the New SUB dialog box, or if nothing appeared at all, you could specify a subprogram name by typing it now. The subprogram name you want (*Chorus*) is there now, so press Enter to open a new subprogram window. You'll see the following:


File Edit View Search Run Debug Options Help
Untitled:Chorus
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00001:011

```
SUB Chorus_
END SUB
```

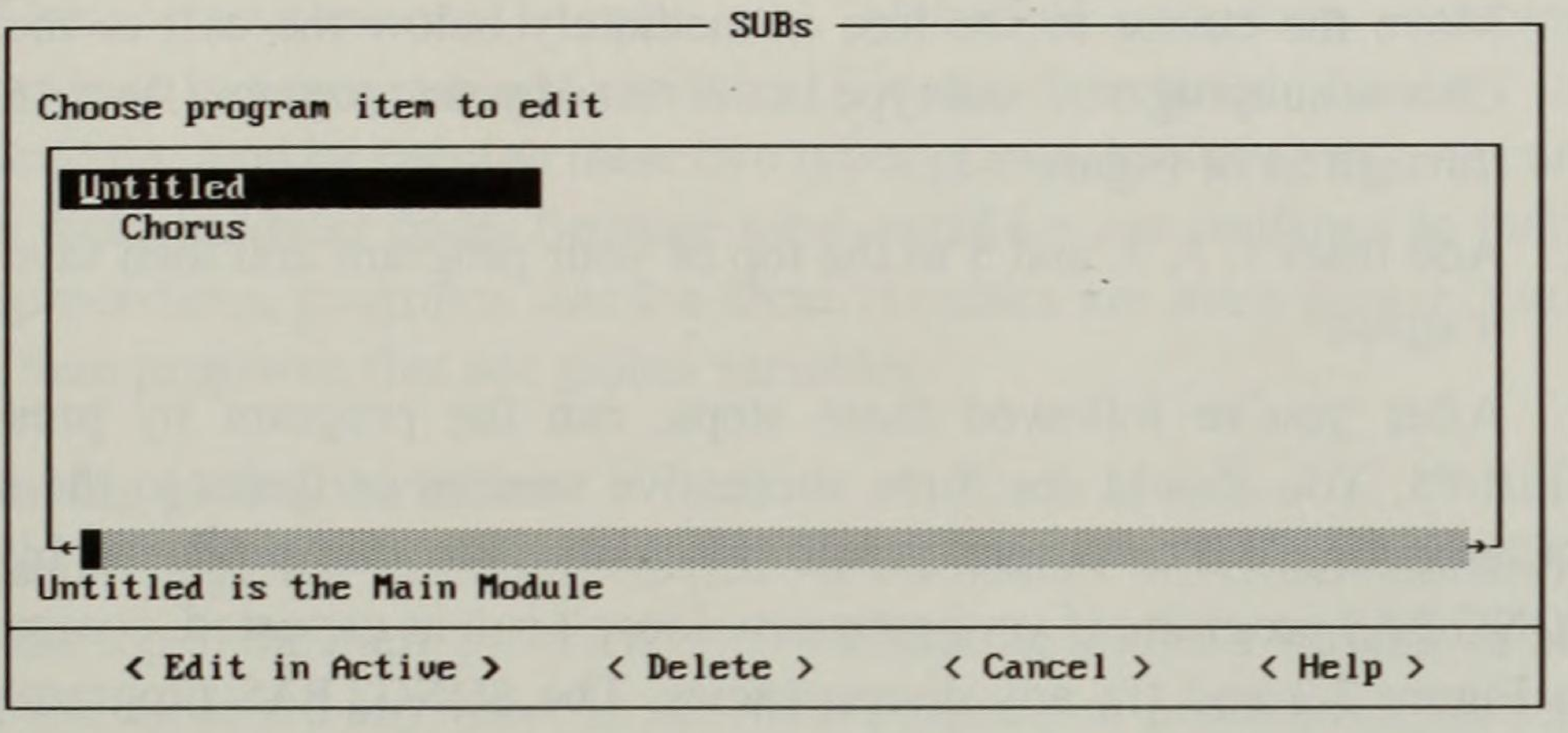
4. There are no parameters to list after the subprogram name, so press Enter to start a new line. Notice that the window title bar now displays the name of the program and the name of the procedure, separated by a colon.

5. Type in the body of the subprogram (lines 54 through 63 of Figure 7-2). Notice that QBasic has added line 64 (the END SUB statement) for you. Your screen should now look like this:


File Edit View Search Run Debug Options Help
Untitled:Chorus
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00011:001

```
SUB Chorus

' The Chorus subprogram prints the chorus of the song "Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"      ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

END SUB
```

6. Select the SUBs command from the View menu to see a dialog box showing the main program, subprograms, and functions in your program.


SUBs

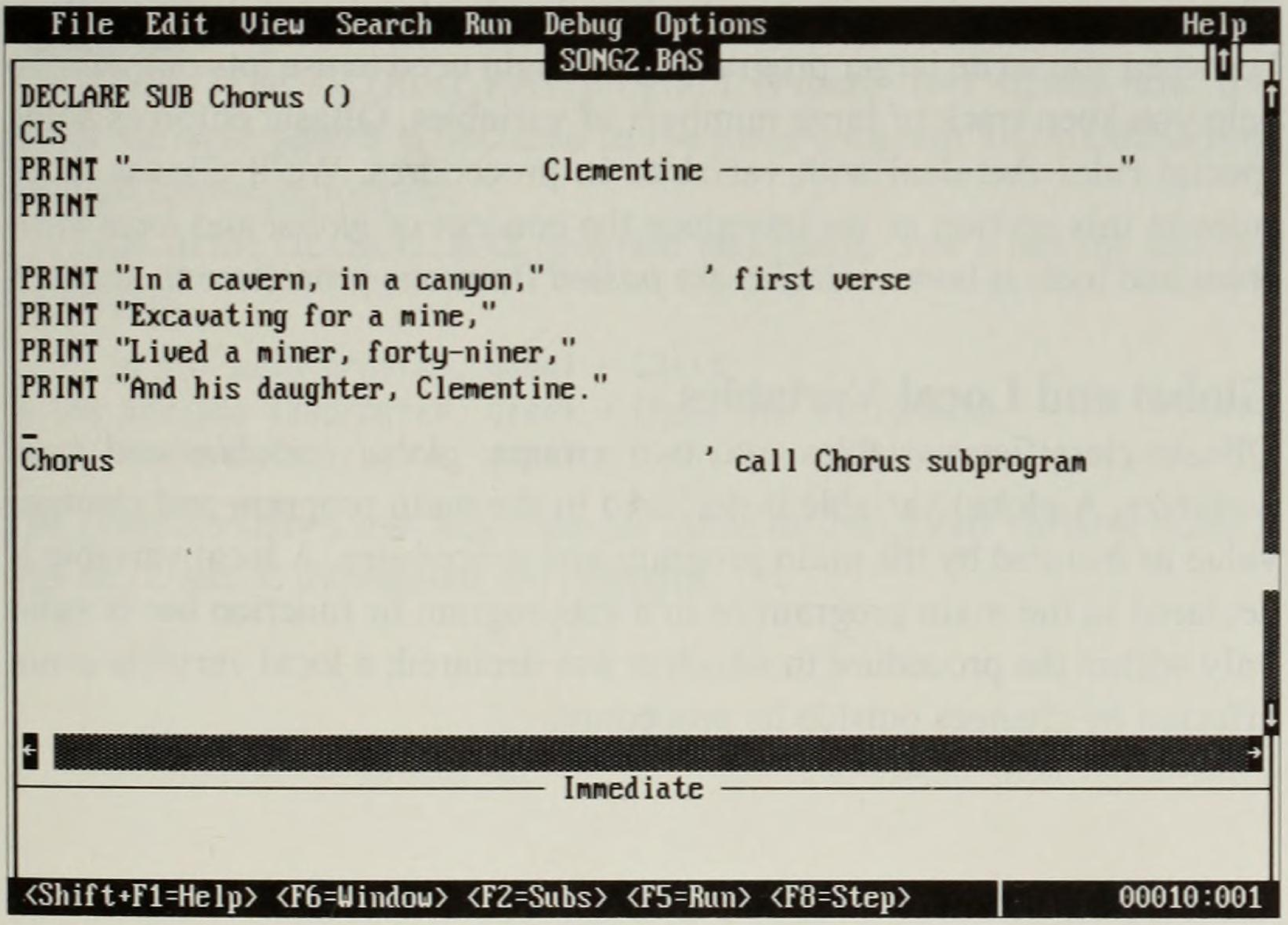
Choose program item to edit

Untitled
Chorus

Untitled is the Main Module

< Edit in Active >
< Delete >
< Cancel >
< Help >

7. Use the direction keys to highlight the main program (currently *Untitled*), and press Enter. QBasic displays the main program in the View window and places the cursor on the line at which you left off.
8. Save your program with the name SONG2.BAS. Before QBasic writes your program to disk, it puts a DECLARE statement at the top of your program (line 4 in Figure 7-2). Your screen should now look like this:


File
Edit
View
Search
Run
Debug
Options
Help
SONG2.BAS

```
DECLARE SUB Chorus ()
CLS
PRINT "----------------------- Clementine -----------------------"
PRINT

PRINT "In a cavern, in a canyon,"              ' first verse
PRINT "Excavating for a mine,"
PRINT "Lived a miner, forty-niner,"
PRINT "And his daughter, Clementine."
_
Chorus                                          ' call Chorus subprogram
```

Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00010:001

9. Move the cursor to the line immediately below the call to the *Chorus* subprogram, and type in the rest of your program (lines 16 through 51 of Figure 7-2).
10. Add lines 1, 2, 3, and 5 to the top of your program and then save it again.

After you've followed these steps, run the program by pressing Shift-F5. You should see three successive screens of lyrics to the song ''Clementine.'' The output should be identical to that produced by the SONG.BAS program. If your program doesn't run as expected, compare it to Figure 7-2 and fix any discrepancies. The SONG2.BAS program is a little longer than the ones we've seen so far in this book, so typing errors might be tougher to find.

This program illustrates a general rule: As your programs get longer, it takes a little more time to get them up and running. Don't be dismayed—this is to be expected. You'll learn more about debugging programs and fixing common programming mistakes in Chapter 12.

USING VARIABLES WITH PROCEDURES

The programs you've written so far have used only a handful of variables, but when you write larger programs you might need to use lots of them. To help you keep track of large numbers of variables, QBasic enforces some special rules that deal with variables in procedures. We'll discuss those rules in this section as we introduce the concept of *global* and *local* variables and look at how variables are *passed* from one procedure to another.

Global and Local Variables

QBasic classifies variables into two groups: *global variables* and *local variables.* A global variable is declared in the main program and changes value as dictated by the main program and procedures. A local variable is declared in the main program or in a subprogram or function but is valid only within the procedure in which it was declared; a local variable is not affected by changes outside its procedure.

These classifications let you use ‘‘globally’’ important variables everywhere in a program and variables of lesser importance ‘‘locally’’ in a procedure. And by keeping these two types of variables distinct, you can write more modular code. Because local variables are confined to their own procedures, programs that use local variables are more general purpose than programs that use global variables.

Declaring global variables

To declare a global variable, you place a COMMON SHARED statement near the top of the main program. This declaration allows a variable to be used and modified anywhere in the main program or procedures. It does not, however, assign a value to the variable. Values are assigned later in the program with assignment statements. The syntax for the COMMON SHARED statement is as follows:

```
COMMON SHARED variableList
```

where *variableList* is a list of one or more variables separated by commas. You can declare any number of variables in one COMMON SHARED statement.

Practice: Using global variables

The GLOBAL.BAS program (Figure 7-4) shows how the global variable *game$* is declared in the main program and modified by the *AddGame* subprogram.

Type in the GLOBAL.BAS program and run it. You’ll see the following output:

```
      In the main program, game$ = Chess
In the AddGame subprogram, game$ = Chess and backgammon
 Back in the main program, game$ = Chess and backgammon
```

The program shows that any changes made to the global variable *game$* will be reflected throughout the program.

```
' GLOBAL.BAS
' This program demonstrates the use of a global variable.

DECLARE SUB AddGame ()     ' declare AddGame subprogram

COMMON SHARED game$        ' declare game$ as a global variable

game$ = "Chess"            ' initialize game$ with value of "Chess"

CLS

PRINT "      In the main program, game$ = "; game$ ' display in main prog.

AddGame                                            ' display in subprogram

PRINT " Back in the main program, game$ = "; game$ ' display in main prog.

END

SUB AddGame

game$ = game$ + " and backgammon"

PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

FIGURE 7-4.
GLOBAL.BAS: A program demonstrating the use of a global variable in a program.

Declaring local variables

Variables are local by default, so you don't have to use any special statements to declare them. Actually, you're an old hand at declaring local variables—you've been doing it all along!

Practice: Using local variables

The LOCAL.BAS program (Figure 7-5) removes the COMMON SHARED statement from the GLOBAL.BAS program to demonstrate that a local variable in the main program and a local variable in a subprogram do not interact with each other. Both variables are named *game$*, and both are string variables. Notice that although both local string variables have the same name, they are completely independent of each other.

Modify the GLOBAL.BAS program so that it matches the LOCAL.BAS program and run it. You'll see the following output:

```
      In the main program, game$ = Chess
In the AddGame subprogram, game$ =  and backgammon
 Back in the main program, game$ = Chess
```

As you can see, the modification of *game$* in the *AddGame* subprogram does not affect the *game$* variable in the main program.

```
' LOCAL.BAS
' This program demonstrates the use of a local variable.

DECLARE SUB AddGame ()     ' declare AddGame subprogram

game$ = "Chess"            ' initialize game$ with value of "Chess"

CLS

PRINT "      In the main program, game$ = "; game$ ' display in main prog.

AddGame                                            ' display in subprogram

PRINT " Back in the main program, game$ = "; game$ ' display in main prog.

END

SUB AddGame

game$ = game$ + " and backgammon"

PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

FIGURE 7-5.
LOCAL.BAS: A program demonstrating the use of a local variable in a program.

Passing Arguments to a Subprogram

You can make any number of variables accessible by declaring them globally. Although convenient, this method increases the chances that your variable will be accidentally modified somewhere in the program.

So QBasic provides a way for you to share arguments (such as variables, constants, and results of expressions and functions) with a limited number of subprograms rather than with the entire program. This method of sharing is called *passing arguments.*

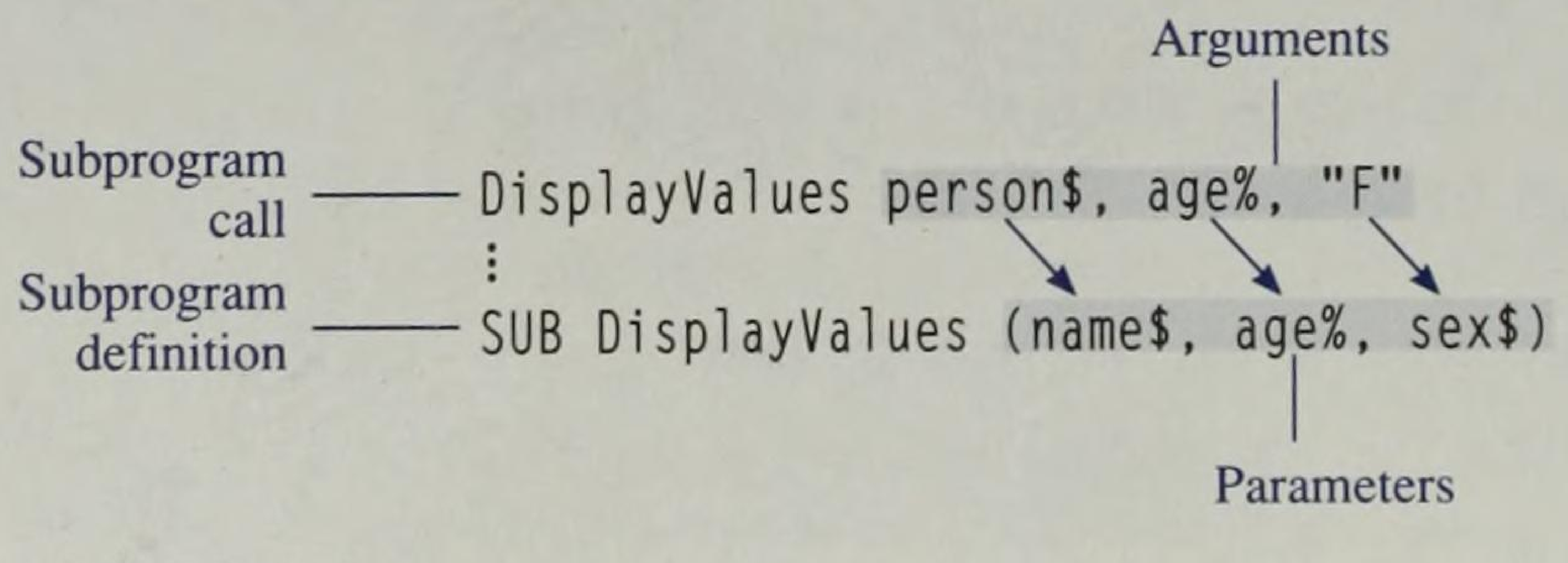
Arguments passed to a subprogram are received by *parameters,* which are local variables within a subprogram. You can use these variables exactly like any other variables.

Arguments vs. parameters

Let's formalize the difference between the terms *argument* and *parameter*:

- **An argument is a constant, a variable, or an expression that is *passed to* a subprogram.** Argument names appear after the subprogram name in a procedure call. A collection of argument names is known as an *argument list.*
- **A parameter is a variable that *receives* a value passed to a subprogram.** Parameter names appear in SUB and DECLARE statements and follow the rules that apply to standard data types. A collection of parameter names is known as a *parameter list.*

Each argument must have a corresponding parameter of the same type (but not necessarily of the same name). Figure 7-6 shows the relationship between an argument list and a parameter list.



```
DisplayValues person$, age%, "F"
:
SUB DisplayValues (name$, age%, sex$)
```

FIGURE 7-6.
The relationship between arguments and parameters.

The following list shows some valid argument-parameter pairs:

```
GetInput name$, number$, address$
⋮
SUB GetInput (person$, phone$, address$)

TranslateToNorsk pig$, dog$
⋮
SUB TranslateToNorsk (gris$, hund$)

PrintHeader TIME$, DATE$, title$, pageNumber%
⋮
SUB PrintHeader (currentTime$, currentDate$, title$, num%)

MixedCall testInt%, testString$, VALUE!, "Hello", 747
⋮
SUB MixedCall (a%, b$, c!, d$, e%)
```

Modifying arguments

Passing arguments to a subprogram is not a one-way street. Any value that a subprogram assigns to a parameter is passed back to the matching argument when the subprogram is finished.

Let's use Figure 7-6 as an example: If *person$* contained the value *Elisabeth* at the time the program called *DisplayValues*, the parameter *name$* would receive that value.

If the subprogram then assigned the value *Vivienne* to *name$*, that value would be assigned to *person$* upon completion of the subprogram.

Keep in mind, however, that constants and expressions are inherently constant: Their values can't change. In Figure 7-6, for example, even if the *DisplayValues* subprogram modified the value of *sex$*, nothing would happen to the matching argument (*"F"*).

Practice: Passing arguments to a subprogram

The ARGUMENT.BAS program (Figure 7-7) demonstrates how two arguments are passed to the *AddInterest* subprogram. The *monthName$* parameter receives the *month$* argument, and the *amount!* parameter receives the *balance!* argument. The *AddInterest* subprogram then changes the month, multiplies the value of *amount!* by 1.05, and returns both values to the main program.

```
' ARGUMENT.BAS
' This program demonstrates passing arguments to a subprogram.

DECLARE SUB AddInterest (monthName$, amount!)

month$ = "January"                ' initialize month$ with value
                                  '   of "January"
balance! = 1500                   ' initialize balance! with value of 1500

CLS

PRINT "Before subprogram:"        ' display original values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!
PRINT

AddInterest month$, balance!      ' call subprogram to modify values

PRINT "After subprogram:"         ' display modified values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!

END

SUB AddInterest (monthName$, amount!)

monthName$ = "February"           ' change name of month
amount! = amount! * 1.05          ' add 5% to amount

END SUB
```

FIGURE 7-7.
ARGUMENT.BAS: A program demonstrating how to pass arguments to a subprogram.

Type in the ARGUMENT.BAS program and run it. You'll see the following output:

```
Before subprogram:
  month$ = January
  balance! = 1500
```

```
After subprogram:
  month$ = February
  balance! = 1575
```

Both *month$* and *balance!* were modified without the use of global variables.

CREATING FUNCTIONS

A function is a block of code placed between FUNCTION and END FUNCTION statements. Functions follow the same general rules as subprograms, with one important exception: A function performs a task and returns a value to the main program or calling procedure. You use a function in the same way that you use a built-in QBasic function—by including it in a QBasic statement. In this section you'll learn how to create your own functions—called *user-defined functions*—with QBasic.

Syntax of a Function

The syntax of a function is as follows:

```
FUNCTION FunctionName (parameterList)
    local variable and constant declarations
    function statements
    FunctionName = returnValue
END FUNCTION
```

- FUNCTION is the QBasic statement that marks the beginning of the function definition.
- *FunctionName* is the name of the function and must end with a type declaration character (just as a variable name can). The function name can be as many as 40 characters long and is the name used to call the function. The function name can't be a QBasic keyword or the same as any variable name or procedure name in your program.
- (*parameterList*) is an optional list of variables.

- *local variable and constant declarations* is an optional list of variables and constants declared and used within the function. They have no effect on variables or constants with the same names elsewhere in the program.
- *function statements* is the working part of the function. You can use almost any QBasic statements in a function.
- *FunctionName* = *returnValue* is an assignment statement in the body of the function (usually near the end) that sets the return value of the function. *returnValue* can be a simple variable or the result of an expression or another function. *FunctionName* and *returnValue* must be of the same type.
- END FUNCTION is the QBasic statement that marks the end of the function definition.

To see how these elements work together, examine the following call, which passes three integer arguments and returns an integer value that is assigned to the *number%* variable:

```
number% = SumOfTerms%(a%, b%, c%)
```

Note that the function name *SumOfTerms%* contains an integer type-declaration character that matches the integer type-declaration character in *number%*. A user-defined function returns a value in one of the five QBasic data types: string, integer, long integer, single-precision floating-point, or double-precision floating-point.

Creating a Function

Creating a user-defined function is quite similar to creating a subprogram:

1. Enter the function using the New Function command from the Edit menu, just as you used the New SUB command to enter a subprogram earlier in the chapter.
2. Examine and edit your functions by using the SUBs command from the View menu. All of your subprograms and functions will be listed together in the SUBs dialog box.

Practice: Using a user-defined function

The AVERAGE.BAS program (Figure 7-8) demonstrates several of the elements of user-defined functions in the declaration of a user-defined function named *Average!*. *Average!* returns the average of three integer parameters passed to the function. After QBasic executes the *Average!* function, the *Average!* name represents a single-precision floating-point value in the main program that can be assigned to a floating-point variable or displayed with a PRINT statement.

```
' AVERAGE.BAS
' This program prints the average of three numbers.

DECLARE FUNCTION Average! (int1%, int2%, int3%)   ' declare function

CLS

PRINT "Enter three integers to be averaged together."
PRINT
INPUT "    First integer:  ", first%
INPUT "    Second integer:  ", second%
INPUT "    Third integer:  ", third%

PRINT
PRINT "The average is"; Average!(first%, second%, third%)

END

FUNCTION Average! (int1%, int2%, int3%)

sum% = int1% + int2% + int3%    ' sum of arguments

Average! = sum% / 3             ' return value is average of arguments

END FUNCTION
```

FIGURE 7-8.
AVERAGE.BAS: A program that prints the value returned by a function.

Type in the AVERAGE.BAS program and run it. You'll see output similar to the following:

```
Enter three integers to be averaged together.

   First integer:  10
   Second integer:  20
   Third integer:  50

The average is 26.66667
```

Similarities to Subprograms

Functions have a number of characteristics in common with subprograms:

- A function call contains the same number of arguments as are present in the function parameter list.
- A function parameter list receives values passed from the main program or calling procedure and assigns them variable names so that they can be used in the body of the function.
- A function can declare and use local variables and constants.
- A function is declared near the top of the main program with a DECLARE statement that is automatically generated by QBasic.
- A function is stored separately in the QBasic environment and is accessible through the SUBs command on the View menu.
- Functions help you modularize your programs and avoid repeated program code by placing often-used routines in easy-to-use modules.

WHICH SHOULD YOU USE?

By now you may be thinking that subprograms and functions are pretty much the same. After all, both subprograms and functions are flexible programming structures that can handle a variety of tasks and return one or more values to the calling module. What, then, are the real differences between subprograms and functions, and which should you use in a given programming situation?

Using the Split Command

The Split command on the View menu lets you view two locations in your program at once. This is particularly useful when your program has many subprograms and functions and you'd like to edit one procedure while looking at another. The following is a list of steps you should follow when you use the Split command. Because you can use the Split command in a number of different ways, you should experiment with Split to see which you like best.

1. Select the Split command from the View menu. The View window will split horizontally into two windows. The upper window, containing the cursor, will be the active window.
2. If you need to make edits in the upper window, scroll to the correct location and make the edits. If both View windows contain the same part of the program, the edits you make in the upper View window will be reflected in the lower View window as soon as you move the cursor off the line you've edited.
3. Press F6 to make the lower View window the active window (or click on it with the mouse). If you would like to move to the upper window again, press F6 twice (or click on the window with the mouse).
4. If you want to place the contents of a procedure into one of the split windows, select the window you want to put the procedure in, press F2 to view the list of procedures, and select the procedure you want to load.
5. When you've finished using split windows, select the Split command from the View menu to switch back to one View window. The active split window will become the active single window.

A Subprogram Is a Miniprogram

Think of a subprogram as a small, self-contained program. A subprogram should complete an important task in a program and yet be general purpose enough to be used in other programming projects. Subprograms are often longer than functions, and are usually used when information is displayed on the screen. The following tasks are often best suited to subprograms:

- Getting input from the user
- Displaying information on the screen
- Processing several numeric values or strings
- Drawing graphic shapes and designs
- Playing musical notes or songs
- Returning multiple values to the calling module

A Function Returns a Value

A function excels in calculating and returning a single value to the main program; functions won't return multiple values or execute general tasks. The following tasks are often best suited to functions:

- Performing a numeric calculation
- Returning a string value
- Generating a random number
- Converting one value to another
- Evaluating a logical expression and returning a value of true or false
- Calculating one result from several arguments

The Main Program Handles Declaration and Control

What tasks are left for the main program to handle? Actually, not many if you make thorough use of subprograms and functions. The following tasks are often best suited to the main program:

- Introductory comments and explanations
- Initialization of key variables and structures

- Program code that is executed only once
- Flow-control structures that determine the path of program execution

The following declarations *must* be included in the main program if they appear in a program:

- Global variable and constant declarations
- Subprogram and function declarations

Now that you've learned about the main program, subprograms, and functions, you're ready to start writing well-organized and modular programs.

SUMMARY

In this chapter you've learned about subprograms and functions—two programming structures that save you from repetition and extend the QBasic language. You define subprograms between SUB and END SUB statements, and you define functions between FUNCTION and END FUNCTION statements. Both kinds of procedures are entered and tracked using the QBasic menu system. Both procedure types support local variables and the exchange of arguments from the calling module. Subprograms and functions, combined with the flow-control and looping structures we discussed in Chapters 5 and 6, make up a complete collection of tools you can use to write structured, well-organized programs.

QUESTIONS AND EXERCISES

1. Which of the following are advantages of programming with procedures?
   a. Procedures can be called any number of times.
   b. Procedures let you create your own statements and functions.
   c. Procedures make program development easier.
   d. General-purpose procedures can be incorporated into other programming projects.

2. What is wrong with the following SUB statement?

   ```
   SUB EnterName$ (firstName$, lastName$)
   ```

3. What is the shortcut function key for the SUBs command on the View menu?

4. What menu command can you use to execute an entire procedure in one step?

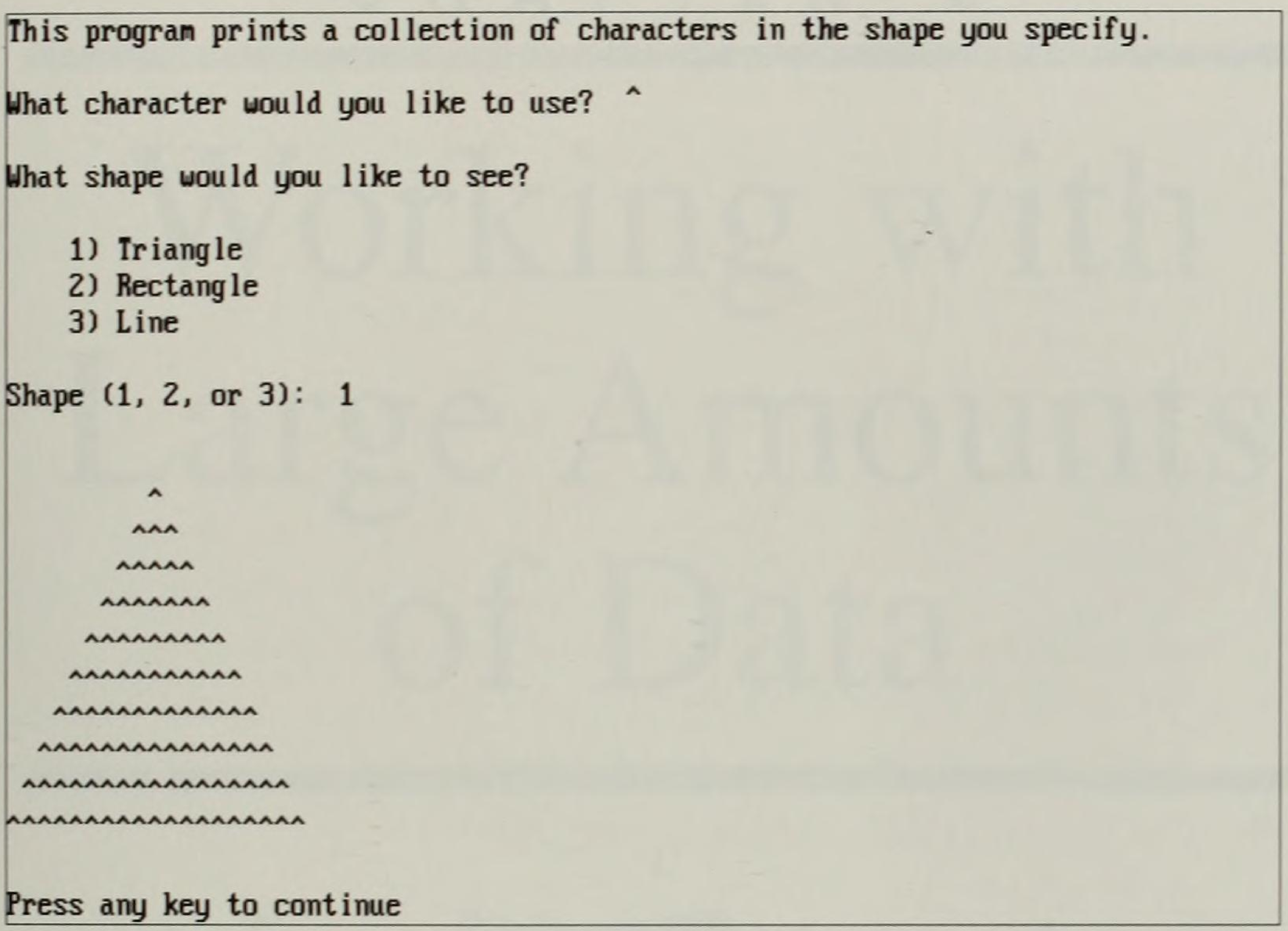
5. Write a QBasic statement that declares a global variable named *total%*.

6. What is the difference between a subprogram and a function?

7. Write a subprogram named *GetCarFacts* that prompts the user for the make, model, year, and color of an automobile and returns this information to the main program in four variables. When you've finished, write a statement that calls the *GetCarFacts* subprogram.

8. Write a function named *FindLarger%* that accepts two integer arguments and returns the larger one to the calling module. When you've finished, write a statement that calls the *FindLarger%* function.

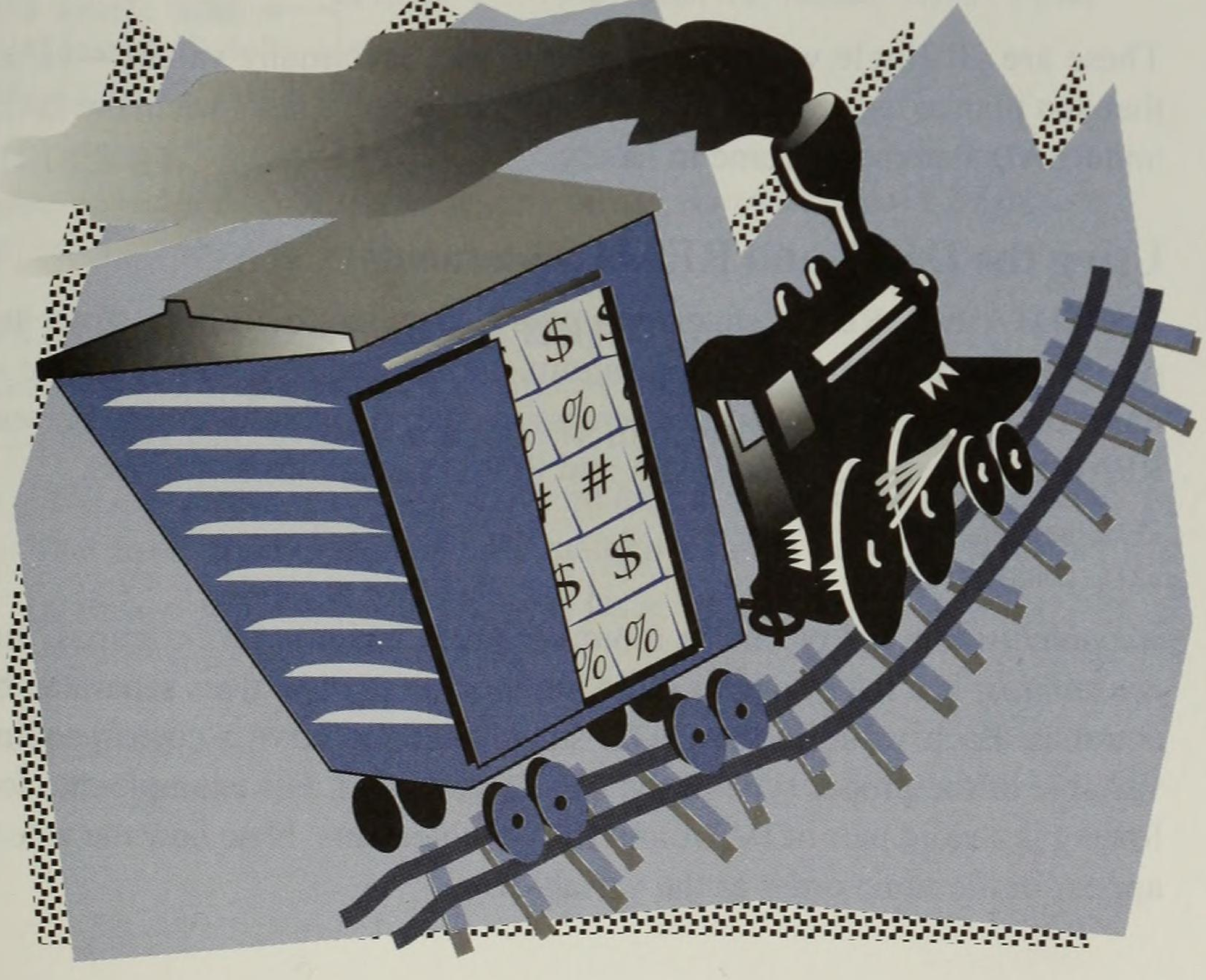
9. Write a program that uses subprograms to print a collection of characters arranged to form one of three shapes: a line, a rectangle, or a triangle. When you run the program, it should produce output similar to the output shown on the next page.



```
This program prints a collection of characters in the shape you specify.

What character would you like to use?  ^

What shape would you like to see?

    1) Triangle
    2) Rectangle
    3) Line

Shape (1, 2, or 3):  1


         ^
        ^^^
       ^^^^^
      ^^^^^^^
     ^^^^^^^^^
    ^^^^^^^^^^^
   ^^^^^^^^^^^^^
  ^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^


Press any key to continue
```

CHAPTER 8

Working with Large Amounts of Data



Now that you've learned how to work with simple data types and variables, it's time to expand your knowledge—to learn how QBasic deals with larger amounts of data in a program. Using the skills you learn in this chapter, you'll be able to create data structures—collections of individual data items—that help you organize large amounts of information and speed up operations involving many variables.

STORING AND RETRIEVING INFORMATION

You've learned three ways to store values in a program:

- Assigning a value to a constant:

  ```
  CONST PRICE! = 12.95
  ```

- Assigning a value to a variable:

  ```
  firstName$ = "Duncan"
  ```

- Assigning a value to a variable with INPUT:

  ```
  INPUT "Enter number of home runs:  ", homers%
  ```

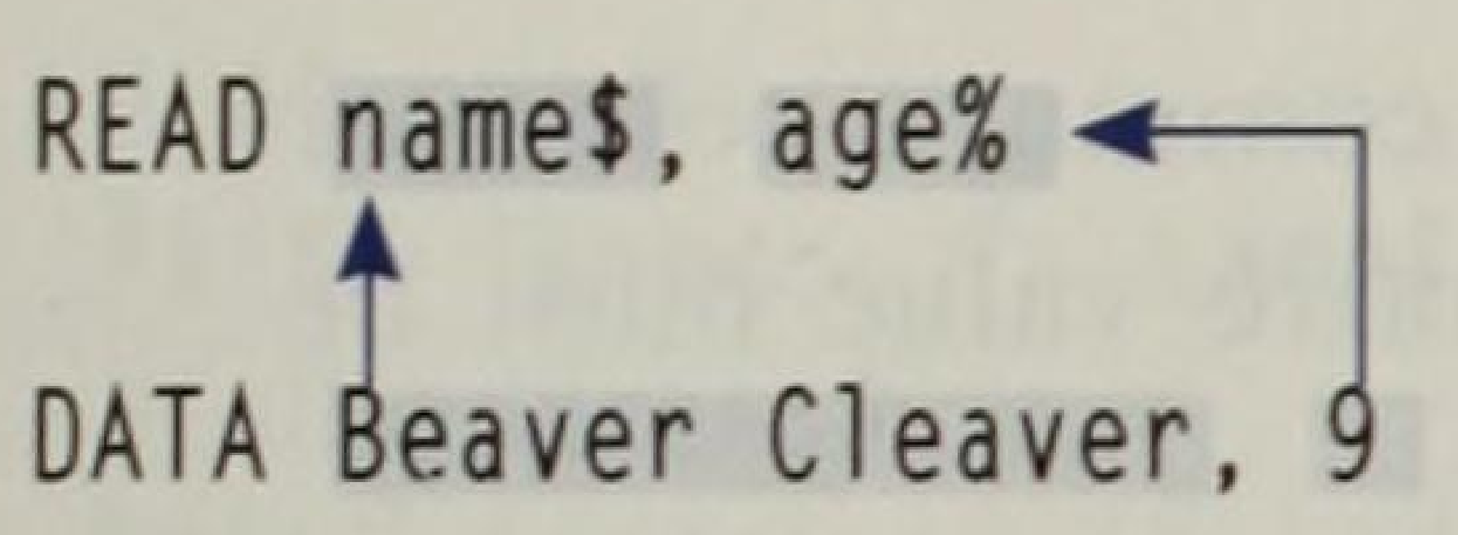
These are all single values. But what if you have many values—values that you plan to use repeatedly, in a specific order? That's when the DATA and READ statements come in handy.

Using the DATA and READ Statements

The DATA and READ statements work together to let you store and retrieve information within a program. Values are first stored in one or more DATA statements and are then assigned to variables with one or more READ statements. Use the DATA and READ statements as follows:

```
READ variableList
DATA constantList
```

variableList is a list of one or more variables separated by commas, and *constantList* is a list of one or more numeric or string values separated by commas. Each value in the DATA statement must have a corresponding variable of the proper type in the READ statement. For example, the following is a valid pair of DATA and READ statements. Note how the values appear in the same order as the variables.



```
READ name$, age%

DATA Beaver Cleaver, 9
```

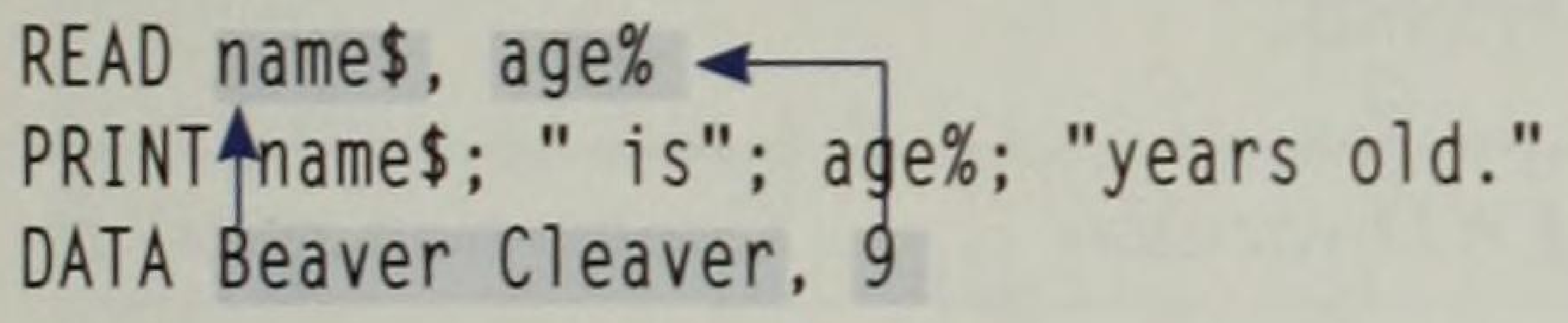
A simple example best demonstrates how DATA and READ work together. In the following program, the READ statement assigns the first DATA value to the string variable *name$* and the second DATA value to the integer variable *age%*. The PRINT statement then makes use of the variables.

```
READ name$, age%
PRINT name$; " is"; age%; "years old."
DATA Beaver Cleaver, 9
```

When you run this program, your output screen displays the following result:

```
Beaver Cleaver is 9 years old.
```

Figure 8-1 shows how data values are assigned to variables.



```
READ name$, age%
PRINT name$; " is"; age%; "years old."
DATA Beaver Cleaver, 9
```

FIGURE 8-1.
Values stored in a DATA statement are assigned to variables with a READ statement.

DATA and READ statements: Helpful hints

Keep the following points in mind as you begin to use the DATA and READ statements to store and retrieve information:

- You can create multiple DATA and READ statements; just be sure that each READ variable has a matching DATA value, as follows:

  ```
  READ month$, numberOfDays%
  READ holidays%

  DATA November
  DATA 30, 3
  ```

- If a value in the DATA statement contains a comma, a colon, or significant trailing or leading spaces, the entire value must be enclosed in quotation marks, as follows:

```
DATA "     Total   ", "Redmond, Washington"
         /         \          /
     Leading    Trailing   Included
      spaces     spaces     comma
```

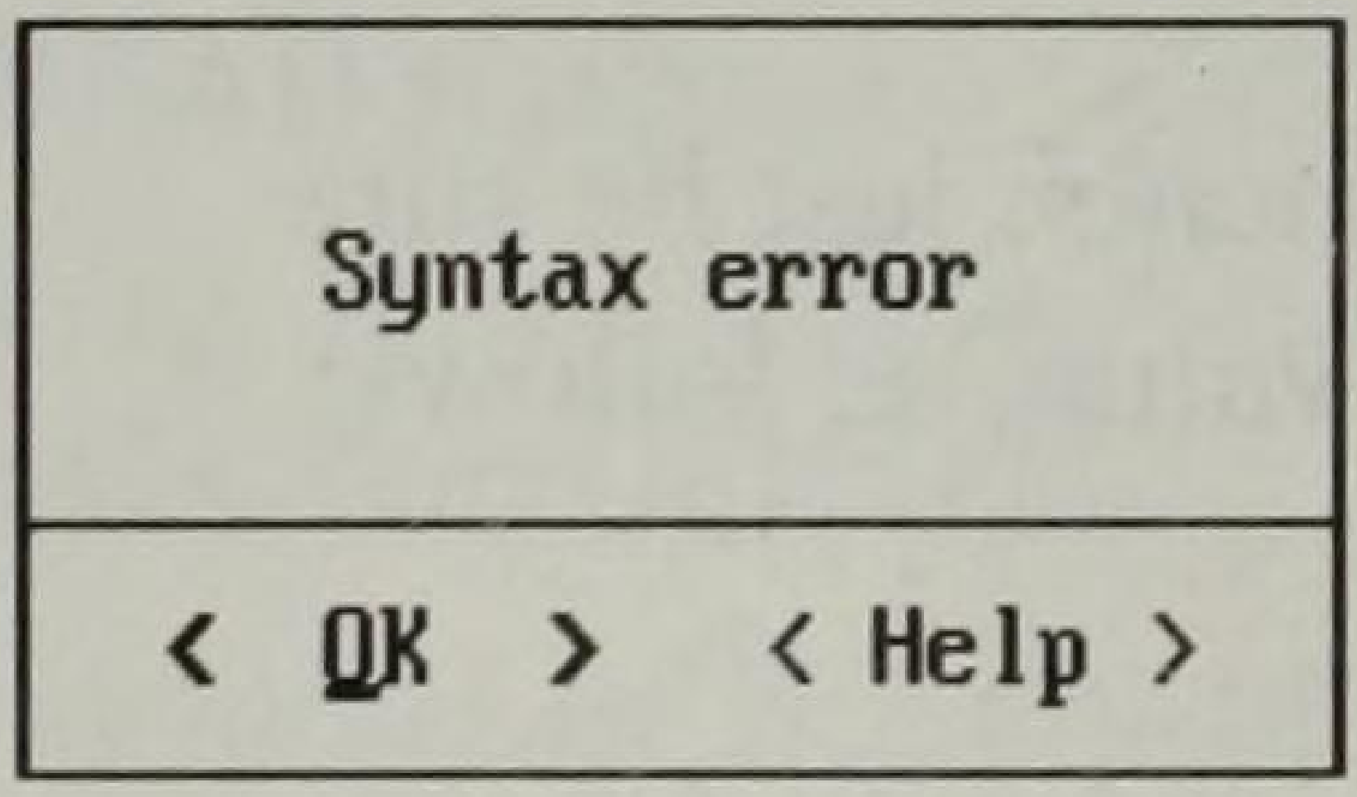
- DATA statements must appear within the main program. (They are commonly placed at the bottom of the main program.) READ statements must be located *above* DATA statements in the main program, but they can appear anywhere in any of the program's procedures.

What if the types don't match?

Each value in a DATA statement must be assigned to a corresponding variable of the same type in a READ statement. If the types don't match, QBasic makes the following conversions:

If DATA value is	*But READ variable is*	*Conversion result is*
string	integer/floating-point	*syntax error*
integer/floating-point	string	string
integer	floating-point	floating-point
floating-point	integer	rounded to integer

A syntax error is indicated by the following dialog box:


Syntax error

The following program demonstrates a type mismatch between a string DATA value and an integer READ variable.

```
READ name$
PRINT "Name:  "; name$
READ address$
PRINT "Address:  "; address$
READ age%
PRINT "Age:  "; age%

DATA Beaver Cleaver, "211 Pine Street, Mayfield", nine
```

When you run the program, your output screen briefly displays the following result:

```
Name:  Beaver Cleaver
Address:  211 Pine Street, Mayfield
```

The *Syntax error* message then appears because the READ statement cannot assign a string value (*nine*) to an integer variable (*age%*). To fix the program, change *nine* to *9* to accommodate the integer variable *age%*, or change *age%* to *age$* to accommodate the string value *nine*.

Typical DATA Statement Values

DATA and READ statements are most appropriate when you know the value of the variables ahead of time and if you know that the values will always appear in the same order. The following values are tailor-made for the DATA and READ method of storage and retrieval:

- Days of the week (Sunday through Saturday)
- Months of the year (January through December)
- Names of persons, places, or organizations
- Numeric data for calculation or analysis
- Numeric values for musical notes

Practice: Storing several values

The ADDTHEM.BAS program (Figure 8-2) shows how to use a READ statement within a FOR loop to assign several DATA values to variables. Note that the integer constant *ITEMS%* controls the number of items that are read from the DATA statements. By changing *ITEMS%*, you can change the number of items to be read.

1. Type in the ADDTHEM.BAS program and run it.

```
' ADDTHEM.BAS
' This program reads and adds values stored in DATA statements.

CONST ITEMS% = 20            ' set the number of items to be read

CLS

FOR i% = 1 TO ITEMS%         ' for each item to be read
    READ number!             '   assign the next DATA item to number!
    sum! = sum! + number!    '   add the item to the running total
NEXT i%

PRINT "The sum of the"; ITEMS%; "numbers is"; sum!

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DATA 88.2, 25, 3.3, 100, -74.2, 0, 20, 0.34, -89, 5.4567
```

FIGURE 8-2.
ADDTHEM.BAS: A program that uses READ to assign 20 data items to variables.

Your output screen displays the following result:

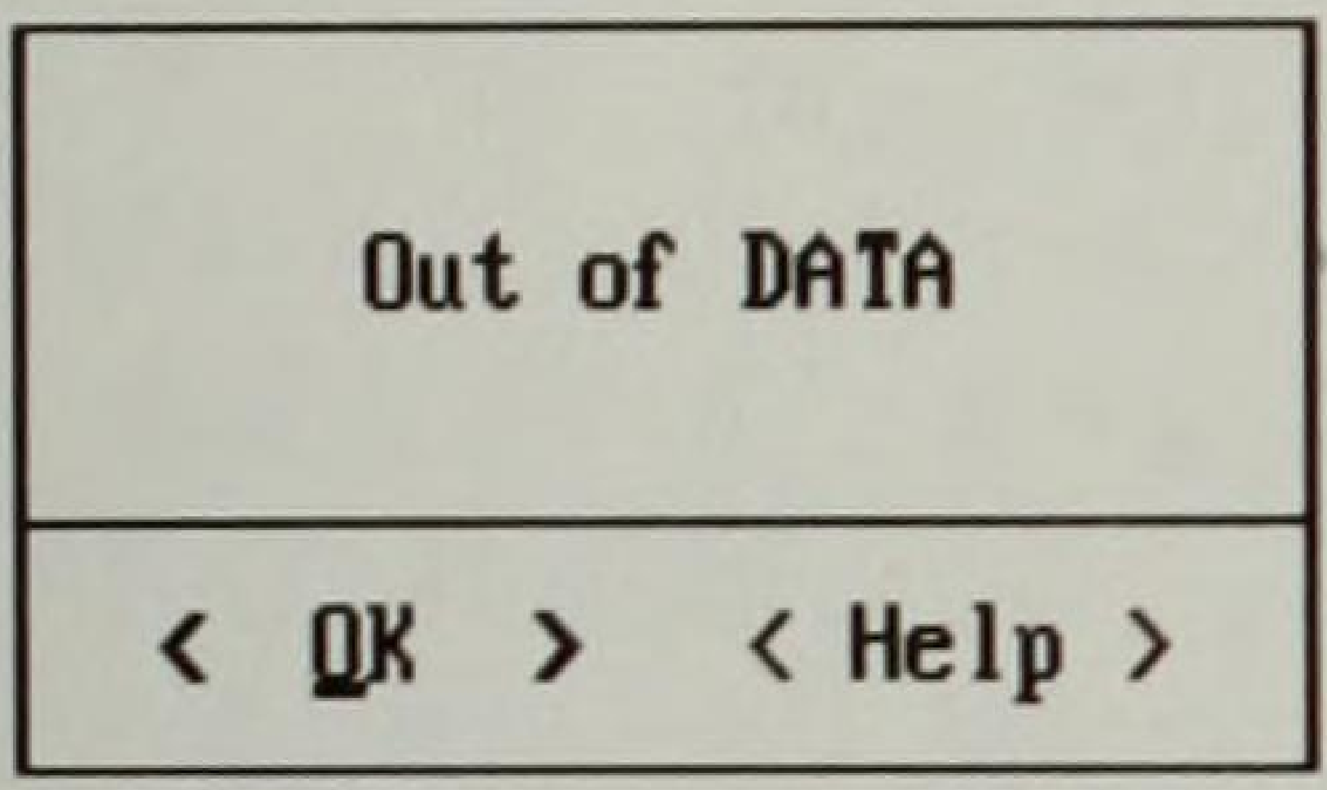
```
The sum of the 20 numbers is 134.0967
```

2. Change the value of the *ITEMS%* constant to *4* and run the program again. Your output screen displays the following result:

```
The sum of the 4 numbers is 10
```

The end-of-data marker

Unless your program can identify the last DATA value, it might keep executing READ statements—trying to assign a value to a variable. The result is the following error message, which indicates that no more data is available for assignment:



Out of DATA

To prevent this error, use an *end-of-data marker* as the final entry in your final DATA statement. The end-of-data marker is a value that your program tests against to determine the end of the DATA values. When it reads this final value, it moves on to execute the rest of the program. This technique is useful when you have a long list of values in DATA statements that will be processed only once. The ADDTHEM.BAS program works around this potential problem by using a constant containing the exact number of values as the upper limit for the FOR loop. Many times you won't have such a luxury. In the following program, ADDTHEM.BAS is revised to check for an end-of-data marker.

Tracking DATA Values: The Data Pointer

To help the READ statement assign values to variables, QBasic uses a *data pointer* to point to the next DATA statement value to be assigned. When a program begins, the data pointer points to the first value in the first DATA statement. As program execution continues, and as DATA values are assigned, the data pointer points to the next unassigned DATA statement value.

Practice: Checking for an end-of-data marker

The ADDTHEM2.BAS program (Figure 8-3) uses an end-of-data marker (–9999) as the last DATA entry in the program.

Type in the ADDTHEM2.BAS program and run it.

```
' ADDTHEM2.BAS
' This program reads and adds values stored in DATA statements
'   until it detects an end-of-data marker (-9999).

CLS

DO WHILE (number! <> -9999)       ' loop until end-of-data marker read
    READ number!                  ' assign next DATA item to number!
    IF (number! <> -9999) THEN    ' if not end-of-data marker then
        sum! = sum! + number!     '   keep a running total
        items% = items% + 1       '   count the number of values read
    END IF
LOOP

PRINT "The sum of the"; items%; "numbers is"; sum!

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DATA 88.2, 25, 3.3, 100, -74.2, 0, 20, 0.34, -89, 5.4567
DATA -9999
```

FIGURE 8-3.
ADDTHEM2.BAS: A program that reads data until an end-of-data marker is reached.

Your output screen displays the following result:

```
The sum of the 20 numbers is 134.0967
```

Rereading DATA values with the RESTORE statement

At times you might want to read repeatedly through the list of DATA values in a program. For example, you might want to perform a number of different calculations on the same set of values. At the point at which you want to return to the first DATA value, use the RESTORE statement, which resets the data pointer to the first DATA statement value in the program. You can use RESTORE as often as you like in a program.

The RESTORE statement has the following syntax:

```
RESTORE
```

Practice: Using RESTORE to repeat a list of values

The TVHOURS.BAS program (Figure 8-4) uses RESTORE, DATA, and READ to track how many hours a person watches television during a three-week period. A pair of nested FOR loops cycles through the days of the week three times (to simulate three weeks), tracking the total number of viewing hours. The RESTORE statement at the end of each cycle returns the data pointer to Monday.

NOTE: To brighten up the output screen, the COLOR statement displays the total number of viewing hours in flashing red. You will see this only if you have a color monitor.

1. Type in the TVHOURS.BAS program and run it. You receive three sets of input prompts similar to this one:

```
How many hours of TV did you watch during the past 3 weeks?

Monday, Week 1 --> 0
Tuesday, Week 1 --> 1
Wednesday, Week 1 --> 0
Thursday, Week 1 --> 2
Friday, Week 1 --> 0
Saturday, Week 1 --> 2.5
Sunday, Week 1 --> 2
```

2. Respond to the prompts. After you enter all 21 values, the output screen displays the final result, which appears in the following form:

```
You watched 23.5 hours of television during the past three weeks!
```

Now that you've learned how to store and retrieve large numbers of values in your program, it's time to learn an efficient way to work with them. The following section introduces the *array*, a powerful data structure that can help you handle large amounts of data of the same type.

```
' TVHOURS.BAS
' This program uses DATA, READ, and RESTORE statements to track
'   the number of TV-viewing hours over a three-week period.

CLS

PRINT "How many hours of TV did you watch during the past 3 weeks?"
PRINT

FOR i% = 1 TO 3       ' for each of the last 3 weeks
    FOR j% = 1 TO 7   '   and for each day in the week
        READ day$                  ' read day name from DATA list
        PRINT day$; ", Week"; i%;  ' prompt with day and week
        INPUT "--> ", hours!       ' get TV hours for that day
        totalHours! = totalHours! + hours!   ' total all the hours
    NEXT j%
    PRINT             ' print a blank line after each week
    RESTORE           ' move data pointer to Monday for next iteration
NEXT i%

PRINT "You watched";               ' display total number of hours
COLOR 20                           '   in blinking red for fun
PRINT totalHours!; "hours";
COLOR 7                            ' return foreground color to white
PRINT " of television during the past three weeks!"

DATA Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
```

FIGURE 8-4.
TVHOURS.BAS: A program demonstrating the RESTORE statement.

WORKING WITH ARRAYS

When you want to organize variables of the same type under one name, use an array. Much as an egg carton organizes a number of individual eggs, an array lets you group many values under one name.

An array can contain any one of the data types you've worked with so far in this book; that is, you can have

- String arrays
- Integer arrays

- Long integer arrays
- Single-precision floating-point arrays
- Double-precision floating-point arrays

You cannot mix types within an array; that is, a string array can contain only strings, an integer array can contain only integers, and so on.

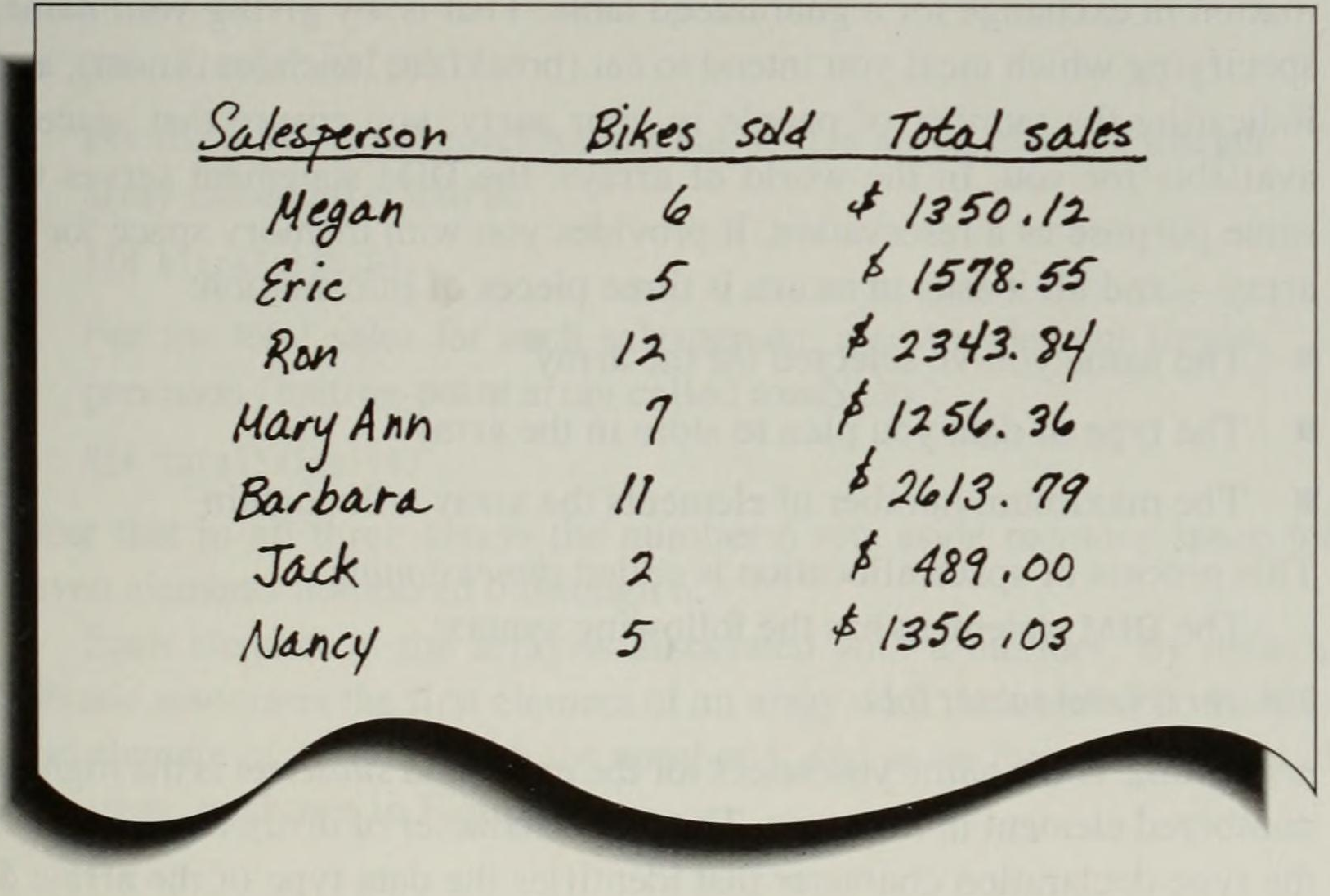
Let's look at an example that organizes data into three types of arrays: string arrays, integer arrays, and single-precision floating-point arrays.

Tracking Information with Arrays: Erin's Bike Market

Erin's Bike Market, a large downtown bicycle shop, has a staff of seven salespersons. Erin wants to track two values every month:

- The number of bikes sold by each salesperson
- The number of dollars brought in by each salesperson

When she writes down this information, it falls naturally into three lists, as shown in Figure 8-5:



Salesperson	Bikes sold	Total sales
Megan	6	$ 1350.12
Eric	5	$ 1578.55
Ron	12	$ 2343.84
Mary Ann	7	$ 1256.36
Barbara	11	$ 2613.79
Jack	2	$ 489.00
Nancy	5	$ 1356.03

FIGURE 8-5.
A sample of salesperson data from Erin's Bike Market.

- A list of string values (salespersons' names)
- A list of integer values (number of bicycles sold by each salesperson)
- A list of floating-point values (total dollar sales for each salesperson)

By making these lists, Erin has already worked with arrays. In fact, she has created three of them: a string array for the salespersons' names, an integer array for the number of bicycles sold by each salesperson, and a single-precision floating-point array for the total dollar sales. Each list, with its distinct type of information, qualifies as an array. And each array contains seven *elements,* or individual values. Now that Erin has organized her arrays on paper, she can begin to convert them into a program with the help of the DIM statement.

The DIM Statement: Making Reservations for Your Array

When you make a reservation at a restaurant, you provide particular information in exchange for a guaranteed table. That is, by giving your name, specifying which meal you intend to eat (breakfast, lunch, or dinner), and indicating the number of people in your party, you ensure that space is available for you. In the world of arrays, the DIM statement serves the same purpose as a reservation. It provides you with memory space for an array—and all it asks in return is three pieces of information:

- The name you've selected for the array
- The type of data you plan to store in the array
- The maximum number of elements the array will contain

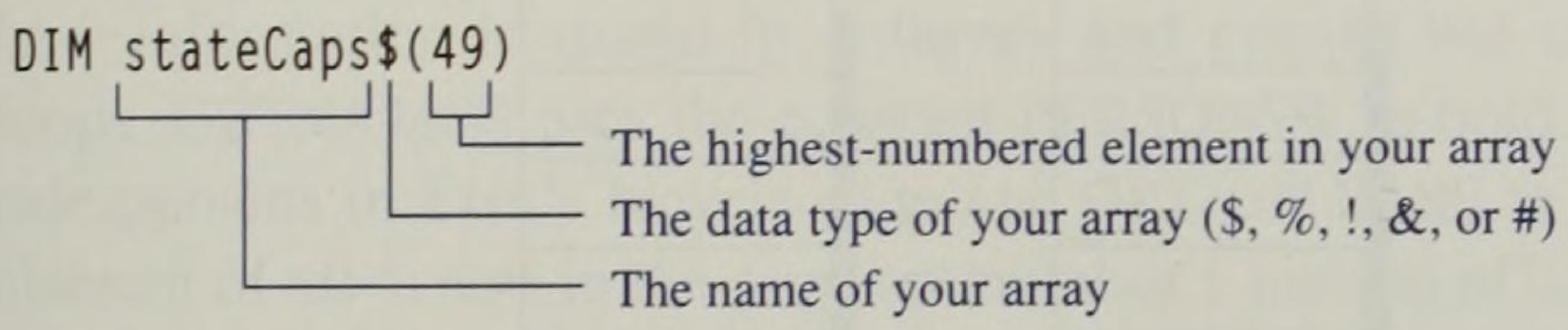
This process of space allocation is called *dimensioning*.

The DIM statement has the following syntax:

```
DIM arrayName(subscript)
```

arrayName is the name you select for the array and *subscript* is the highest numbered element in the array. The final character of *arrayName* must be the type-declaration character that identifies the data type of the array: *$* for strings, *%* for integers, *&* for long integers, *!* for single-precision floating-point numbers, or # for double-precision floating-point numbers.

The following DIM statement, for example, dimensions a string array that can contain up to 50 elements numbered 0 through 49:



```
DIM stateCaps$(49)
```

The highest-numbered element in your array

The data type of your array ($, %, !, &, or #)

The name of your array

When you execute this DIM statement, QBasic reserves memory space for a 50-element array to contain the names of the state capitals.

NOTE: A typical computer has room for many arrays, each containing thousands of elements. But because computer memory is not unlimited, you shouldn't set aside memory space for more array elements than you think you'll need.

Erin would create the three arrays she needs by using these DIM statements:

- For the names of the salespersons, a seven-element string array called *salesGroup$*:

  ```
  DIM salesGroup$(6)
  ```

- For the number of bicycles each has sold, a seven-element integer array called *bikesSold%*:

  ```
  DIM bikesSold%(6)
  ```

- For the total sales for each salesperson, a seven-element single-precision floating-point array called *totalSales!*:

  ```
  DIM totalSales!(6)
  ```

Note that in all three arrays the number *6* sets aside memory space for seven elements numbered 0 through 6.

Each element in the array is associated with a number. By default, QBasic associates the first element of an array with the number 0, the second element of an array with the number 1, and so on for each element of the array, as shown in Figure 8-6.

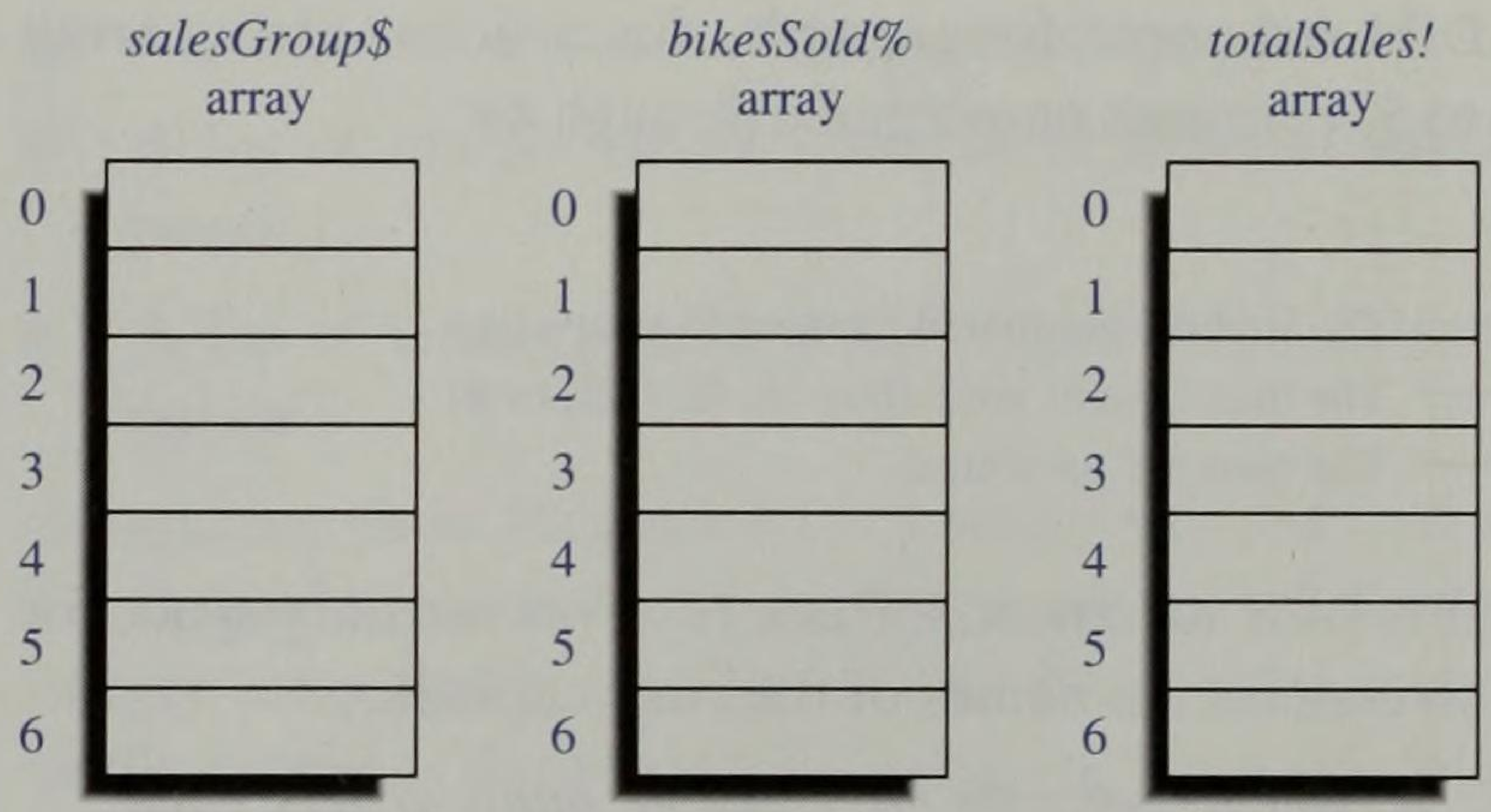


FIGURE 8-6.
Each element of a dimensioned array is associated with a number.

Working with Array Elements

After you've dimensioned an array with the DIM statement, it's easy to refer to any of the elements within the array. To refer to an element of an array, you use the array name and an array *index* surrounded by parentheses. The index must be an integer value; for example, it can be a simple number or an integer variable. The following statement assigns the string value *Tomato* to element 3 in the *shoppingList$* array:

```
shoppingList$(3) = "Tomato"
```

The OPTION BASE Statement

If you think your program would be clearer conceptually if the first element in each array were 1 instead of 0, use the OPTION BASE statement. The OPTION BASE statement associates the first element—or *base*—of all arrays in a program with the number 1. To use OPTION BASE, simply place the following statement near the top of your program, before any DIM statements:

```
OPTION BASE 1
```

The programs throughout this chapter use the OPTION BASE statement this way.

Practice: Storing values in an array

The GETNAMES.BAS program (Figure 8-7) demonstrates how information is stored in an array and printed out using two FOR loops. GETNAMES uses the constant PERSONS% to hold the number of salespersons in Erin's bicycle shop and OPTION BASE to make the first element of all arrays in the program number 1 instead of 0.

Type in the GETNAMES.BAS program and run it.

```
' GETNAMES.BAS
' This program reads string information into an array.

CONST PERSONS% = 7            ' number of salespersons
OPTION BASE 1                 ' set base of all arrays to 1

DIM salesGroup$(PERSONS%)     ' dimension salesGroup$ string array

CLS

FOR i% = 1 TO PERSONS%        ' use i% to access array elements
    INPUT "Enter salesperson name:  ", salesGroup$(i%)
NEXT i%

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO PERSONS%        ' print entire contents of array
    PRINT salesGroup$(i%)
NEXT i%
```

FIGURE 8-7.
GETNAMES.BAS: A program demonstrating loading and printing an array.

You'll be prompted to supply names for the salespeople. You'll receive output similar to the following:

```
Enter salesperson name:  Megan
Enter salesperson name:  Eric
Enter salesperson name:  Ron
Enter salesperson name:  Mary Ann
```

(continued)

Formatting Your Output: The PRINT USING Statement

The PRINT USING statement lets you design your output's appearance on the screen. PRINT USING is particularly helpful if you need to display large amounts of data in tabular form. To create the design you want, you use a *template* string. The syntax for the PRINT USING statement is as follows:

```
PRINT USING template; argumentList
```

template is a string and *argumentList* is a collection of one or more values to be displayed, separated by semicolons. *template* specifies how the values in *argumentList* should be displayed. The formatting characters in *template* must match up one for one with the characters of the values in *argumentList*. The following table describes a few of the useful formatting characters that can be included in *template*:

Character(s)	***Description***
#	Represents one digit of a numeric value
.	Represents the decimal point in a numeric value
$$	Cause a dollar sign to be displayed with a number
\ \	Represent one or more spaces that can be filled with string data

The following program creates a formatting template named *tmp$* and uses it in a PRINT USING statement to display a string variable and a dollars-and-cents single-precision floating-point value:

```
tmp$ = "Name:  \             \  Payment: $$####.##"
name$ = "Duncan Hund"
payment! = 2496.33
PRINT USING tmp$; name$; payment!
```

When you execute the statements, you see this output:

```
Name:  Duncan Hund      Payment:  $2496.33
```

continued

```
Enter salesperson name:  Barbara
Enter salesperson name:  Jack
Enter salesperson name:  Nancy

You entered the following names:

Megan
Eric
Ron
Mary Ann
Barbara
Jack
Nancy
```

Practice: Using multiple arrays in a program

The BIKEINFO.BAS program (Figure 8-8) demonstrates how a number of arrays can be used together in a program to track related information. BIKEINFO is a revision of the GETNAMES program. This version tracks the salesperson names with the *salesGroup$* array, the number of bikes each person sold with the *bikesSold%* array, and the total value of each person's sales with the *totalSales!* array. The three arrays are designed to be used together—each contains a piece of information about the salespersons in the bike shop, as shown in Figure 8-6. If you refer to the array indexes together in a FOR loop, your program can access related items at the same time. The PRINT USING statement and the template *tmp$* display the data in each array.

Type in the BIKEINFO.BAS program and run it.

```
' BIKEINFO.BAS
' This program reads information into three arrays and prints it.

CONST PERSONS% = 7            ' number of salespersons
OPTION BASE 1                 ' set base of all arrays to 1
```

FIGURE 8-8. *(continued)*
BIKEINFO.BAS: A program demonstrating the use of three arrays in a FOR loop.

FIGURE 8-8. *continued*

```
DIM salesGroup$(PERSONS%)    ' dimension salesGroup$ string array
DIM bikesSold%(PERSONS%)     ' dimension bikesSold% integer array
DIM totalSales!(PERSONS%)    ' dimension totalSales! floating-point array

CLS

FOR i% = 1 TO PERSONS%       ' get salesperson name and sales data
    INPUT "Enter salesperson name:  ", salesGroup$(i%)
    INPUT "   Bikes sold:  ", bikesSold%(i%)
    INPUT "   Total sales:  $", totalSales!(i%)
    PRINT
NEXT i%

PRINT "You entered the following sales data:"
PRINT
PRINT "Salesperson     Bikes sold     Total sales"
PRINT "------------------------------------------"
PRINT

' initialize tmp$, a formatting template for PRINT USING
tmp$ = "\              \ ###           $$####.##"

FOR i% = 1 TO PERSONS%       ' print contents of each array
    PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
NEXT i%
```

You'll receive output similar to the following:

```
Enter salesperson name:  Megan
   Bikes sold:  6
   Total sales:  $1350.12

Enter salesperson name:  Eric
   Bikes sold:  5
   Total sales:  $1578.55

Enter salesperson name:  Ron
   Bikes sold:  12
   Total sales:  $2343.84
```

```
Enter salesperson name:  Mary Ann
   Bikes sold:  7
   Total sales:  $1256.36

Enter salesperson name:  Barbara
   Bikes sold:  11
   Total sales:  $2613.79

Enter salesperson name:  Jack
   Bikes sold:  2
   Total sales:  $489

Enter salesperson name:  Nancy
   Bikes sold:  5
   Total sales:  $1356.03

You entered the following sales data:

Salesperson     Bikes sold     Total sales
------------------------------------------

Megan                6           $1350.12
Eric                 5           $1578.55
Ron                 12           $2343.84
Mary Ann             7           $1256.36
Barbara             11           $2613.79
Jack                 2            $489.00
Nancy                5           $1356.03
```

Filling Part of an Array

You needn't fill an array to the brim. In fact, allowing a little room for future growth is often a good idea. If you intend to do this, however, you need a way to signal QBasic that the user has finished entering data. One way to do this is to use an end-of-data marker, as discussed earlier in the chapter. You can have the user add elements to the array as part of a WHILE loop that continually checks for the end-of-data marker. As soon as QBasic sees the marker, it exits the loop and executes the rest of the program. The end-of-data marker should be the same data type as the array being filled, and it should be a value that the user is unlikely to type during the normal execution of the program. A typical end-of-data marker for a string array is *QUIT* or *END*. A typical end-of-data marker for a numeric array is *–9999*.

Practice: Using an end-of-data marker

The DATAEND.BAS program (Figure 8-9) demonstrates how to fill an array with different amounts of data. The program fills and prints three arrays again, but this time dimensions the arrays with 50 elements each and uses an end-of-data marker of *END* to indicate that the user is finished entering data. (Note that *END* must be entered in all capital letters.)

Type in the DATAEND.BAS program and run it.

```
' DATAEND.BAS
' This program reads information into three arrays and prints it.
'   The maximum number of names that can be entered is 50; fewer
'   can be entered by typing "END" for the salesperson name.

OPTION BASE 1                   ' set base of all arrays to 1

DIM salesGroup$(50)             ' dimension salesGroup$ string array
DIM bikesSold%(50)              ' dimension bikesSold% integer array
DIM totalSales!(50)             ' dimension totalSales! floating-point array

CLS

PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                      ' initialize an array counter variable

WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)
```

FIGURE 8-9. *(continued)*

DATAEND.BAS: A program that uses an end-of-data marker to determine when input is complete.

FIGURE 8-9. *continued*

```
    IF (salesGroup$(count%) <> "END") THEN
        INPUT "   Bikes sold:  ", bikesSold%(count%)
        INPUT "   Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1             ' increment the array counter
    END IF
WEND

PRINT
PRINT "You entered the following sales data:"
PRINT
PRINT "Salesperson     Bikes sold     Total sales"
PRINT "------------------------------------------"
PRINT

' initialize tmp$, a formatting template for PRINT USING
tmp$ = "\               \ ###          $$####.##"

FOR i% = 1 TO count% - 1     ' print contents of each array
    PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
NEXT i%
```

You'll receive output similar to the following:

```
Follow prompts to enter bike shop data.  Type END to quit.

Enter salesperson name:  Nancy
   Bikes sold:  5
   Total sales:  $1356.03

Enter salesperson name:  Barbara
   Bikes sold:  11
   Total sales:  $2613.79

Enter salesperson name:  END

You entered the following sales data:

Salesperson     Bikes sold     Total sales
------------------------------------------

Nancy               5           $1356.03
Barbara            11           $2613.79
```

Creating Flexible Arrays

As you've learned, when you use the DIM statement you must inform QBasic of the number of elements in your array so that it can reserve an adequate amount of space in memory. But what if you're not sure how many elements your array will contain? For example, if your program is dependent on user input for the contents of the array, the number of elements entered might vary each time the program is run. How can you tell QBasic how much memory to reserve?

The DIM statement is actually quite accommodating. In fact, it can create two kinds of arrays depending on the type of information you provide: *static arrays* and *dynamic arrays.*

Static arrays

A static array is an array of a specific size—a size that you know in advance. For example, in the BIKEINFO.BAS program, the number of salespersons is known in advance and placed into the program as a constant:

```
CONST PERSONS% = 7
```

The DIM statement then uses the constant PERSONS% as the number of elements in the array:

```
DIM salesGroup$(PERSONS%)
```

Dynamic arrays

A dynamic array is an array whose size can change. A dynamic array is dimensioned only when the user of the program defines the number of elements the array is to contain. To create a dynamic array, you would follow these steps:

1. Use the INPUT statement to prompt the user for the number of elements.
2. Assign the value entered by the user to an integer variable.
3. Use the integer variable with the DIM statement to dimension the array.

Use a static array when you know in advance how big the array should be. Use a dynamic array when the array size will be determined anew each time the program is run.

Practice: Using a variable to set array size

The DYNAMIC.BAS program (Figure 8-10) uses the *persons%* variable to dimension the three dynamic sales arrays. Note the IF statement that checks the value of *persons%*: If *persons%* is less than or equal to 0, no arrays are dimensioned.

Type in the DYNAMIC.BAS program and run it.

```
' DYNAMIC.BAS
' This program reads information into three dynamic arrays and prints it.

OPTION BASE 1                   ' set base of all arrays to 1

CLS

INPUT "How many salesperson names would you like to enter?  ", persons%
IF (persons% > 0) THEN          ' must be at least one salesperson

    DIM salesGroup$(persons%)   ' dimension salesGroup$ string array
    DIM bikesSold%(persons%)    ' dimension bikesSold% integer array
    DIM totalSales!(persons%)   ' dimension totalSales! floating-pt. array

    PRINT

    FOR i% = 1 TO persons%      ' get salesperson name and sales data
        INPUT "Enter salesperson name:  ", salesGroup$(i%)
        INPUT "   Bikes sold:  ", bikesSold%(i%)
        INPUT "   Total sales:  $", totalSales!(i%)
        PRINT
    NEXT i%

    PRINT "You entered the following sales data:"
    PRINT
```

FIGURE 8-10. *(continued)*
DYNAMIC.BAS: A program demonstrating the use of three dynamic arrays.

FIGURE 8-10. *continued*

```
    PRINT "Salesperson     Bikes sold     Total sales"
    PRINT "------------------------------------------"
    PRINT

    ' initialize tmp$, a formatting template for PRINT USING
    tmp$ = "\               \ ###          $$####.##"

    FOR i% = 1 TO persons%     ' print contents of each array
        PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
    NEXT i%

END IF
```

You'll receive output similar to the following:

```
How many salesperson names would you like to enter?  2

Enter salesperson name:  Megan
   Bikes sold:  6
   Total sales:  $1350.12

Enter salesperson name:  Eric
   Bikes sold:  5
   Total sales:  $1578.55

You entered the following sales data:

Salesperson     Bikes sold     Total sales
------------------------------------------

Megan               6           $1350.12
Eric                5           $1578.55
```

Searching for an Element in an Array

At times you might want to perform a search within an array. You generally do this to find a specific array element or to find an array element based on comparison. Both operations involve stepping through an array one element at a time and keeping track of matches to a search string.

- In a search for a specific array element, QBasic compares a search string with each element of the array until a match is found or until all array elements have been examined.
- In a comparison search, you use one or more temporary variables to track the progress of the comparison. Comparisons usually take one of the following forms:
    - ☐ Find the largest number in the array.
    - ☐ Find the smallest number in the array.

The following exercises demonstrate typical methods for finding a specific array element and for finding the largest number in an array.

Practice: Finding an array element

The SEARCH.BAS program (Figure 8-11) demonstrates how to search an array for a specific element. The program prompts the user for salesperson data and then asks which salesperson's data the user would like to examine. A FOR loop steps through each element of the *salesGroup$* array until a match is found or all the array entries have been examined:

- If a match is found, the corresponding elements from the *bikesSold%* and *totalSales!* arrays are displayed, and then the loop is exited with an EXIT FOR statement.

The EXIT FOR Statement

The EXIT FOR statement lets you exit a FOR loop early—before the loop finishes on its own. EXIT FOR is useful when you want to loop a specific number of times *unless* a certain condition is met. The following program asks for names until the user has entered 10 names or until the user enters *QUIT*, whichever occurs first:

```
FOR i% = 1 TO 10
    INPUT "Enter a name:  ", firstName$
    IF (firstName$ = "QUIT") THEN EXIT FOR
    PRINT firstName$
NEXT i%
```

- If no match is found, the message *Name not found* is displayed.

  Type in the SEARCH.BAS program and run it.

```
' SEARCH.BAS
' This program reads data into three arrays and searches for a name.
'   The maximum number of names that can be entered is 50; fewer
'   can be entered by typing "END" for the salesperson name.

OPTION BASE 1                ' set base of all arrays to 1

DIM salesGroup$(50)          ' dimension salesGroup$ string array
DIM bikesSold%(50)           ' dimension bikesSold% integer array
DIM totalSales!(50)          ' dimension totalSales! floating-point array

CLS
PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                   ' initialize an array counter variable
WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)

    IF (salesGroup$(count%) <> "END") THEN
        INPUT "   Bikes sold:  ", bikesSold%(count%)
        INPUT "   Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1  ' increment the array counter
    END IF
WEND

PRINT                        ' prompt user for search string
INPUT "What name would you like to search for?  ", search$
PRINT

' initialize tmp$, a formatting template for PRINT USING
tmp$ = "\              \ ###         $$####.##"
```

FIGURE 8-11. *(continued)*

SEARCH.BAS: A program demonstrating how an array can be searched for a specific element.

FIGURE 8-11. *continued*

```
' compare each array element with search string until a match is
'   found, then display the record and exit the loop; display
'   message if search string is not found

FOR i% = 1 TO count% - 1      ' count% - 1 is the last array element
    IF (salesGroup$(i%) = search$) THEN
        PRINT "Salesperson     Bikes sold     Total sales"
        PRINT "-----------------------------------------"
        PRINT
        PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
        EXIT FOR              ' this statement breaks out of a FOR loop
    END IF
    IF (i% = count% - 1) THEN PRINT "** Name not found **"
NEXT i%
```

You'll receive output similar to the following:

```
Follow prompts to enter bike shop data.  Type END to quit.

Enter salesperson name:  Jack
   Bikes sold:  2
   Total sales:  $489.00

Enter salesperson name:  Ron
   Bikes sold:  12
   Total sales:  $2343.84

Enter salesperson name:  Mary Ann
   Bikes sold:  7
   Total sales:  $1256.36

Enter salesperson name:  END

What name would you like to search for?  Ron

Salesperson     Bikes sold     Total sales
------------------------------------------


Ron                 12           $2343.84
```

Practice: Finding the largest number in an array

The MAXSALES.BAS program (Figure 8-12) demonstrates how the largest element in an array can be extracted by use of a FOR loop. The program prompts the user for salesperson data and then examines each element of the *totalSales!* array. The largest sales figure is stored in the variable *largest!* and is compared with each element of *totalSales!*. If an array element is larger than *largest!*, that array element becomes the new *largest!*. (The *lg%* variable stores the array index associated with *largest!*.) After it examines all array elements, the program displays the largest sales figure and the related elements in the *salesGroup$* and *bikesSold%* arrays.

Type in the MAXSALES.BAS program and run it.

```
' MAXSALES.BAS
' This program reads salesperson data into three arrays and displays
'   data for the salesperson with the highest total sales.  The maximum
'   number of names that can be entered is 50; fewer can be entered by
'   typing "END" for the salesperson name.

OPTION BASE 1                   ' set base of all arrays to 1

DIM salesGroup$(50)             ' dimension salesGroup$ string array
DIM bikesSold%(50)              ' dimension bikesSold% integer array
DIM totalSales!(50)             ' dimension totalSales! floating-point array

CLS

PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                      ' initialize an array counter variable

WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)
```

FIGURE 8-12. *(continued)*
MAXSALES.BAS: A program demonstrating the extraction of the largest element in an array.

FIGURE 8-12. *continued*

```
    IF (salesGroup$(count%) <> "END") THEN
        INPUT "  Bikes sold:  ", bikesSold%(count%)
        INPUT "  Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1  ' increment the array counter
    END IF
WEND

largest! = totalSales!(1)    ' first array element is largest so far
lg% = 1                      ' save array index

' compare remaining array elements for something bigger--if one is
'   found, assign it to largest! and save the array index in lg%; if
'   there is a tie for the largest, return the first element found

FOR i% = 2 TO count% - 1
    IF (totalSales!(i%) > largest!) THEN
        largest! = totalSales!(i%)  ' save new largest value
        lg% = i%                    ' save array index
    END IF
NEXT i%

' initialize tmp$, a formatting template for PRINT USING
tmp$ = "\               \ ###          $$####.##"

PRINT
PRINT "** "; salesGroup$(lg%); " has the highest total sales **"
PRINT
PRINT "Salesperson     Bikes sold     Total sales"
PRINT "----------------------------------------"
PRINT
PRINT USING tmp$; salesGroup$(lg%); bikesSold%(lg%); totalSales!(lg%)
```

You'll receive output similar to the following:

```
Follow prompts to enter bike shop data.  Type END to quit.

Enter salesperson name:  Megan
   Bikes sold:  6
   Total sales:  $1350.12

Enter salesperson name:  Eric
   Bikes sold:  5
   Total sales:  $1578.55

Enter salesperson name:  Nancy
   Bikes sold:  5
   Total sales:  $1356.03

Enter salesperson name:  Mary Ann
   Bikes sold:  7
   Total sales:  $1256.36

Enter salesperson name:  END

** Eric has the highest total sales **

Salesperson     Bikes sold     Total sales
------------------------------------------

Eric                5            $1578.55
```

Two-Dimensional Arrays

QBasic also lets you declare arrays of two dimensions. A two-dimensional array represents a table of values with rows and columns, such as a scoreboard, an accounting ledger, or a gameboard. In this section we'll discuss how to declare and use a two-dimensional array in a QBasic program.

Let's start with an example. Sam, the local soda distributor, wants to track sales for his top four brands over the last 12 months. He wants to put the information in a table that uses brand names for row titles and months for column titles, as shown in Figure 8-13.

	Jan	*Feb*	*Mar*	*Apr*	*May*	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*Nov*	*Dec*
Orca Spray	64	63	58	45	36	32	41	39	50	67	69	103
Fizzy Delite	35	41	60	57	38	29	25	19	26	37	43	36
Alki Seltzer	15	9	12	21	24	32	46	42	37	22	18	13
Schpritz	30	30	30	35	42	44	49	48	38	35	31	30

FIGURE 8-13.
A table of values showing soda sales over the last 12 months.

Using Arrays with Procedures

In Chapter 7 you learned how variables can be declared locally or globally and how variable arguments are passed to subprograms and functions. A similar set of rules applies to arrays used in programs that contain procedures:

- By default, an array is local to the main program or procedure it is declared in.
- An array is declared global with the COMMON SHARED statement in the main program; for example:

```
DIM stores$(20)
COMMON SHARED stores$()
```

- A single array element can be passed to a procedure; for example:

```
GetInput stores$(12)
```

- An entire array can be passed to a procedure by specifying no subscript; for example:

```
PrintAllStores stores$()
```

As the programs you write become larger, you'll want to use arrays right along with subprograms and functions to keep your code organized and efficient.

Tabular information is perfectly suited to a two-dimensional array. In this example, one dimension of the array corresponds to the brand name rows and the other dimension corresponds to the month columns. Items in a two-dimensional array are identified with row *and* column subscripts. Figure 8-14 shows how row and column subscripts would be assigned to each dimension if the base of the array were set at 1. A one-dimensional array requires one subscript to identify an array element, and a two-dimensional array requires two subscripts to identify an array element. In Figure 8-14, for example, the number of cases of Orca Spray sold in May would be identified by row 1, column 5.

Row subscripts

Column subscripts

		1	2	3	4	5	6	7	8	9	10	11	12
1	**Orca Spray**	64	63	58	45	36	32	41	39	50	67	69	103
2	**Fizzy Delite**	35	41	60	57	38	29	25	19	26	37	43	36
3	**Alki Seltzer**	15	9	12	21	24	32	46	42	37	22	18	13
4	**Schpritz**	30	30	30	35	42	44	49	48	38	35	31	30

FIGURE 8-14.
Assigning two-dimensional array subscripts to the soda sales table.

Declaring a two-dimensional array

To dimension a two-dimensional array, use the DIM statement as follows:

```
DIM arrayName(rows, columns)
```

arrayName is the name of the array, *rows* is the number of rows in the array (the first dimension), and *columns* is the number of columns in the array (the second dimension). *rows* and *columns* must be integers and can be expressed as numbers, constants, or variables. The final character of *arrayName* must be the type-declaration character that identifies the data type of the array: *$* for a string, *%* for an integer, *&* for a long integer, *!* for a single-precision floating-point array, or # for a double-precision floating-point array. As in a one-dimensional array, every element in a two-dimensional array must be of the same data type.

NOTE: The first element in each dimension of the array is numbered 0 unless you use the OPTION BASE 1 statement before you dimension the array.

The following statements dimension an array named *sodaSales%* with 4 rows and 12 columns:

```
OPTION BASE 1
DIM sodaSales%(4, 12)
```

These statements ask the user to enter the dimensions of the dynamic two-dimensional array named *sodaSales%*:

```
OPTION BASE 1
INPUT "Enter number of soda brands sold:  ", brands%
INPUT "Enter number of months to track:  ", months%
DIM sodaSales%(brands%, months%)
```

Practice: Building a table of values

The SALES.BAS program (Figure 8-15) demonstrates how the two-dimensional array *sodaSales%* is filled and displayed on the screen. SALES gets the numbers of rows and columns in the soda sales table from the user, stores those numbers in the *brands%* and *months%* integer variables, and uses them to dimension the *sodaSales%* array. SALES uses two nested FOR loops to fill and display the array and uses DATA and READ statements to store and retrieve the months of the year. Use this program as a guide for filling and printing any two-dimensional dynamic array.

Type in the SALES.BAS program.

```
' SALES.BAS
' This program tracks the sales of soda over a given number of months.

CLS

PRINT "** Soda sales tracking program **"
PRINT
DO
   INPUT "How many brands of soda do you sell?  ", brands%
LOOP WHILE (brands% < 1)
```

FIGURE 8-15. *(continued)*

SALES.BAS: A program that uses a dynamic two-dimensional array to track soda sales.

FIGURE 8-15. *continued*

```
DO
    INPUT "How many months would you like to record (1-12)?  ", months%
LOOP WHILE (months% < 1) OR (months% > 12)
PRINT

OPTION BASE 1                     ' set first array element at 1
DIM sodaSales%(brands%, months%)  ' dimension soda sales array
DIM brandNames$(brands%)          ' dimension brand name array

' get names of soda brands sold

PRINT "Enter the"; brands%; "brands of soda you sell"
PRINT
FOR i% = 1 TO brands%
    INPUT "Brand name:  ", brandNames$(i%)
NEXT i%

' get soda sales for each month

PRINT
PRINT "Enter soda sales in cases"
PRINT

FOR i% = 1 TO brands%           ' for each brand of soda...
    PRINT "** "; brandNames$(i%); " **"
    PRINT                       '   print name of brand
    FOR j% = 1 TO months%       ' for each month...
        READ mo$                '   read month name from DATA list
        PRINT "  "; mo$;        '   print month name and prompt for input
        INPUT ":  ", sodaSales%(i%, j%)  ' store input in array
    NEXT j%
    PRINT
    RESTORE                     ' rewind DATA list to first month
NEXT i%

' print out soda sales table

CLS
```

(continued)

FIGURE 8-15. *continued*

```
PRINT "Soda sales in cases for"; months%; "months"
PRINT
PRINT "---------------------------------------------------------------"
PRINT "Brand/Month   ";

FOR i% = 1 TO months%
    PRINT " ";
    READ mo$                  ' read month name from DATA list
    PRINT mo$;                ' print month names across top of table
NEXT i%

PRINT
PRINT "---------------------------------------------------------------"

' templates for PRINT USING

nameTmp$ = "\            \"    ' for brand name (up to 12 digits)
salesTmp$ = " ###"            ' for cases sold (up to 3 digits)

FOR i% = 1 TO brands%         ' fills in the table
    PRINT USING nameTmp$; brandNames$(i%);
    FOR j% = 1 TO months%
        PRINT USING salesTmp$; sodaSales%(i%, j%);
    NEXT j%
    PRINT
NEXT i%

PRINT "---------------------------------------------------------------"

' data for READ statements above

DATA Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
```

When you run SALES.BAS, you are prompted for input:

```
** Soda sales tracking program **

How many brands of soda do you sell? 4
How many months would you like to record (1-12)? 12
```

(continued)

continued

```
Enter the 4 brands of soda you sell

Brand name:  Orca Spray
Brand name:  Fizzy Delite
Brand name:  Alki Seltzer
Brand name:  Schpritz

Enter soda sales in cases

** Orca Spray **

  Jan:  64
  Feb:  63
  Mar:  58
  ⋮
```

After the data for each brand name and month has been entered, you'll receive output similar to the following:

```
Soda sales in cases for 12 months

--------------------------------------------------------------
Brand/Month    Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
--------------------------------------------------------------
Orca Spray      64  63  58  45  36  32  41  39  50  67  69 103
Fizzy Delite    35  41  60  57  38  29  25  19  26  37  43  36
Alki Seltzer    15   9  12  21  24  32  46  42  37  22  18  13
Schpritz        30  30  30  35  42  44  49  48  38  35  31  30
--------------------------------------------------------------
```

NOTE: To print a copy of the final table on your printer, be sure that your printer is on, and then hold down the Shift key and press the PrtSc key. The Shift-PrtSc key combination sends all the text on the screen to the printer.

Array Troubleshooting

Although arrays are a great boon to your programs, they also increase the potential for error. Let's look at some typical programming errors associated with arrays and with processing large amounts of data and then look at some ways to avoid the errors.

Mistake 1: Not using an integer to define a subscript

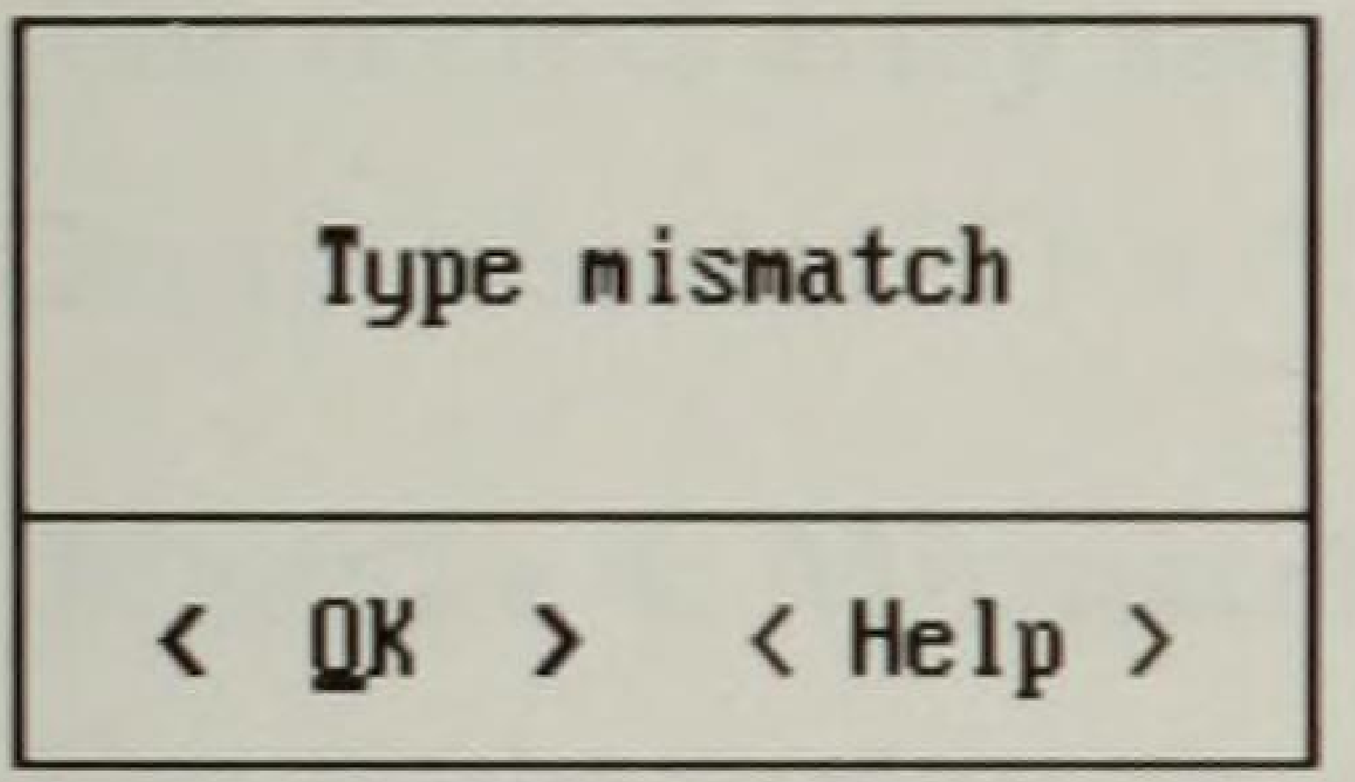
The number of elements and dimensions in an array is determined by integer subscripts in the DIM statement. *Be sure that the array subscripts are integer values or variables.* The following array declaration, for example, is valid:

```
DIM grades%(students%, 15)
```

The subscripts *students%* and *15* are both integer values. The following declaration is not valid because *items$* and *quantity$* are string values:

```
DIM costTable!(items$, quantity$)
```

When you try to dimension an array with invalid subscripts, you receive the following error message:



Type mismatch

If you supply a floating-point number as a subscript, QBasic rounds the value to the nearest integer value and then dimensions the array. For example, the following statement successfully dimensions an array with 6 elements (0 through 5):

```
DIM values%(4.6)
```

NOTE: Using a floating-point value as an array subscript will be confusing to a person reading your program. Stick to integer subscripts.

Mistake 2: Using a DIM statement in a loop

Be sure that DIM statements are outside any loops in your program. The following program fragment demonstrates the incorrect use of a DIM statement to declare a dynamic array:

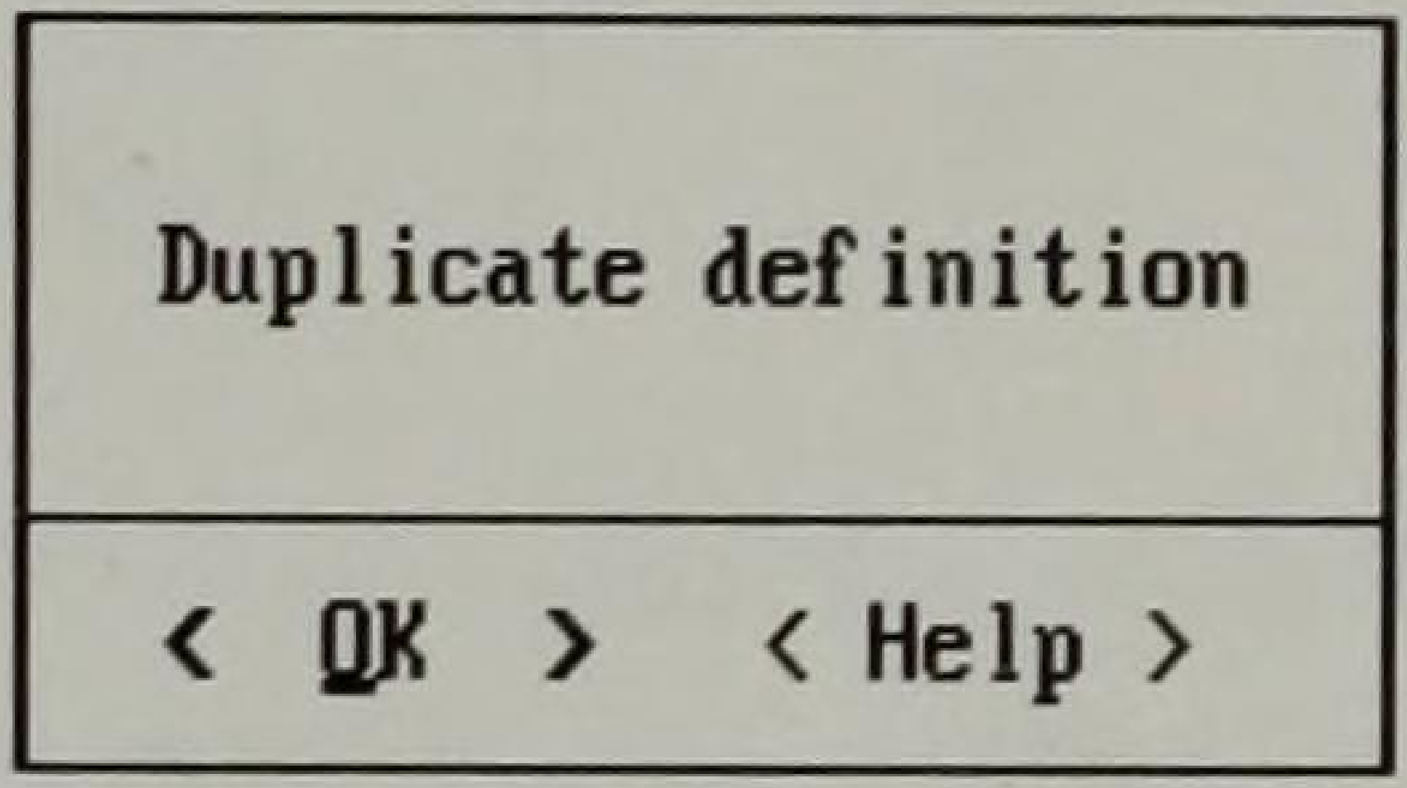
```
OPTION BASE 1
size% = 5
CLS
```

(continued)

continued

```
FOR i% = 1 to size%
    DIM names$(size%)
    INPUT "Enter a name:  ", names$(i%)
NEXT i%

PRINT
FOR i% = 1 to size%
    PRINT names$(i%)
NEXT i%
```

If you run the program, you receive the following error message when the DIM statement is executed a second time:



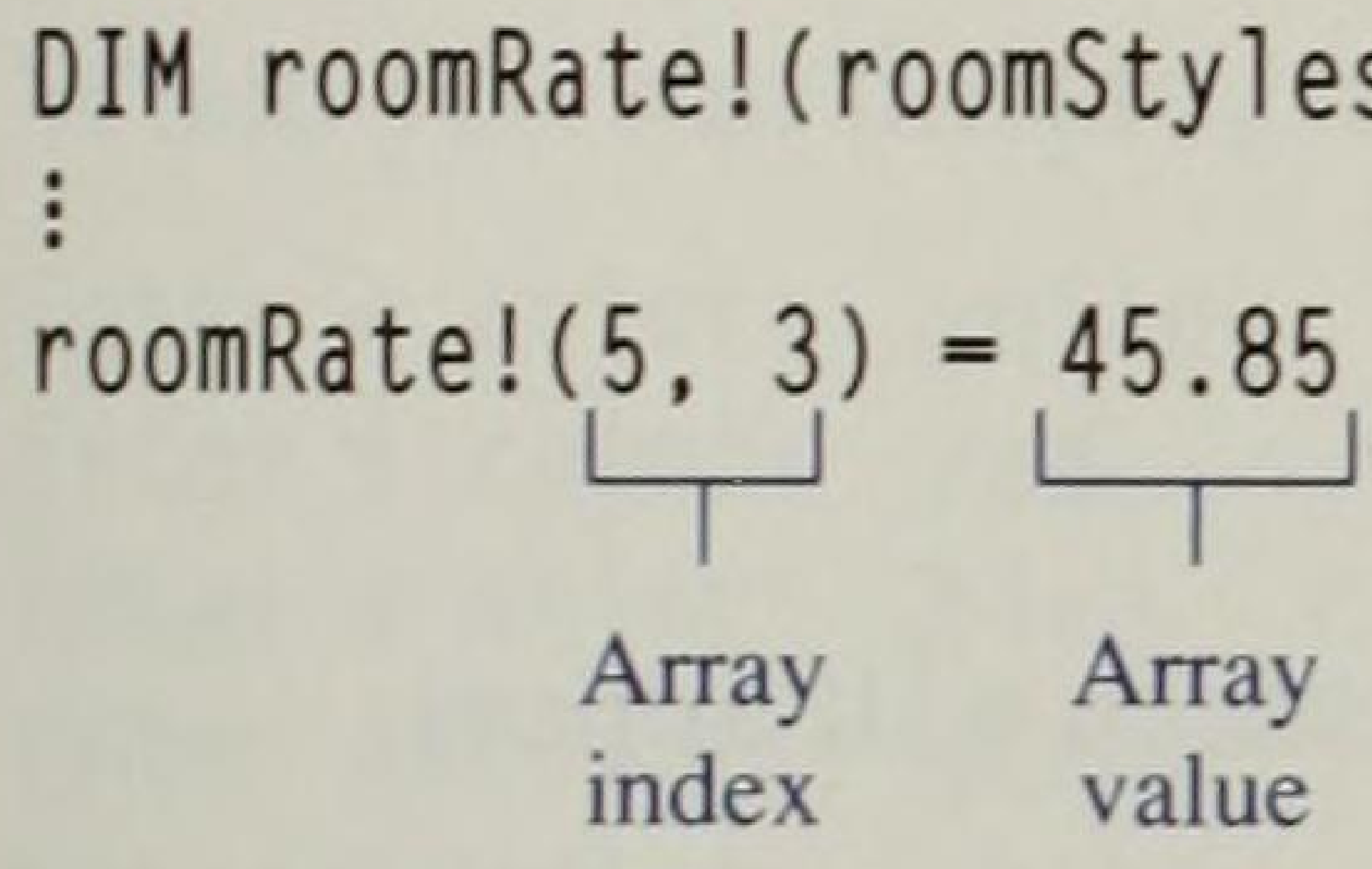
Duplicate definition

To avoid this error, dimension the *names$* array before the loop, as follows:

```
OPTION BASE 1
size% = 5
DIM names$(size%)
CLS

FOR i% = 1 to size%
    INPUT "Enter a name:  ", names$(i%)
NEXT i%

PRINT
FOR i% = 1 to size%
    PRINT names$(i%)
NEXT i%
```

Mistake 3: Confusing the array index with the array value

It's easy to confuse the value used to *index* an array element with the value *in* the array element. Remember that the array index is always in parentheses and that it follows the array name that describes the location of the array value (Figure 8-16).



```
DIM roomRate!(roomStyles%, daysRented%)
:
roomRate!(5, 3) = 45.85
```

FIGURE 8-16.
Array index and array value.

The following statement tries to assign the string value ''Orcas Hotel'' to the fifth element in the *vacationSpots$* array but fails because the array index and the array value are in the wrong locations:

```
vacationSpots$("Orcas Hotel") = 5
```

The correct array assignment is as follows:

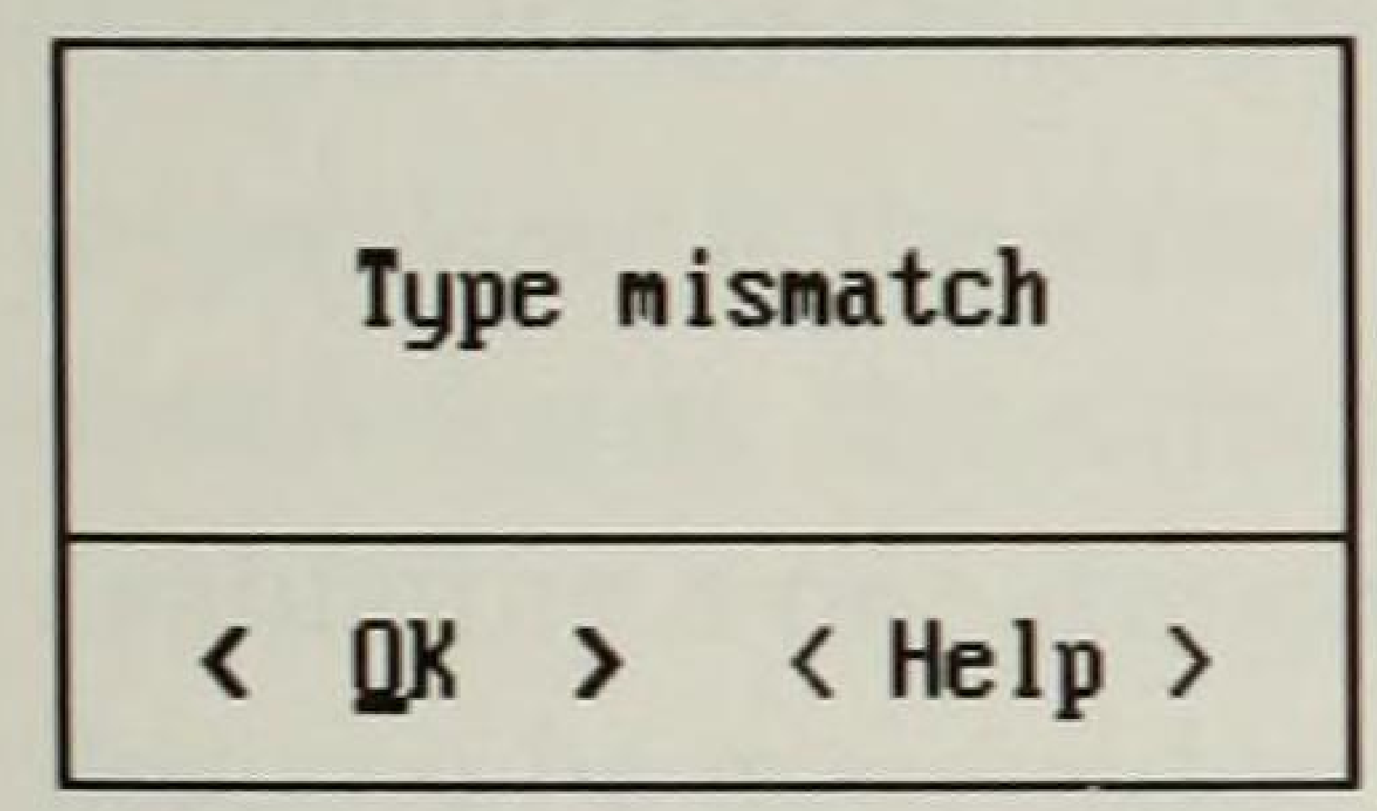
```
vacationSpots$(5) = "Orcas Hotel"
```

Mistake 4: Mismatching array types

Every element in an array must be of the same data type. If you try to assign a variable or value to an array that does not match the array type specified in the DIM statement, you'll receive an error message. For example, the following assignment statement generates an error message because *cost!* is a single-precision floating-point array and the value *"57.36"* is a string data type. (A correct assignment would be *57.36* without the quotation marks.)

```
cost!(i%) = "57.36"
```

This kind of assignment error produces the following message:



Type mismatch

A similar type mismatch occurs while the program is running if the user enters a value that does not match the array element that receives the value. The following loop, for example, prompts the user to enter several integers.

```
DIM values%(5)
FOR i% = 1 TO 5
    INPUT "Enter a number:  ", values%(i%)
NEXT i%
```

If the user enters a string value, the following message will appear:

```
Enter a number:  ten

Redo from start
Enter a number:
```

Redo from start is QBasic's way of prompting the user for data of the correct type. Usually, this message is enough to get users back on track. As we've mentioned before, however, the best way to avoid input problems is to spell out with your input prompt exactly what you want.

Mistake 5: Making out-of-range errors

Attempting to refer to an element that does not exist in an array generates an *out-of-range* error message when the program is run. For example, QBasic generates an error message when the third of the following statements is executed because *testScores%* can contain only 25 elements and a reference is made to a 30th element.

```
OPTION BASE 1
DIM testScores%(25)
testScores%(30) = 92
```

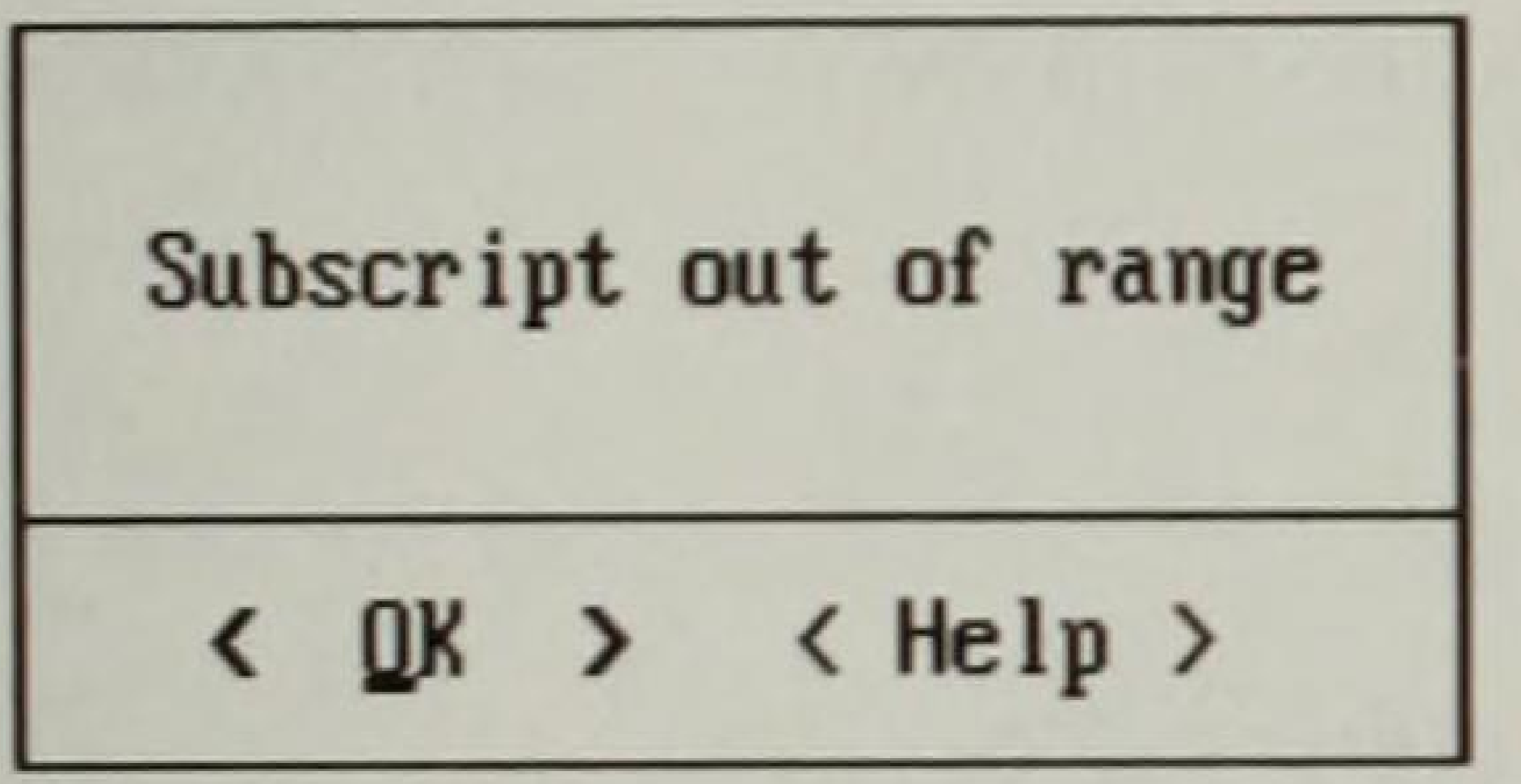
The out-of-range error is most common in a looping structure, as shown in the following program fragment:

```
OPTION BASE 1
DIM colors$(4)
count% = 0
CLS

DO
    count% = count% + 1
    INPUT "Enter a color name:  ", colors$(count%)
LOOP UNTIL (colors$(count%) = "QUIT")
```

When you run the program fragment, you see output something like this:

```
Enter a color:  yellow
Enter a color:  red
Enter a color:  blue
Enter a color:  green
Enter a color:  white
```

After the fifth color is entered (filling the last element in the array), the *count%* loop counter is incremented and the INPUT statement (now referencing an element beyond the end of the array) triggers the *Subscript out of range* error message.



Subscript out of range

The solution for out-of-range problems is to determine the upper and lower bounds of the array and not go past them. QBasic provides the UBOUND and LBOUND functions to return the bounds of any array in your program so that you can handle this potential problem.

The UBOUND and LBOUND functions

The UBOUND and LBOUND functions return integer values corresponding to the upper and lower bounds of an array. Here's the syntax for the UBOUND and LBOUND functions:

```
UBOUND(arrayName, dimension)
LBOUND(arrayName, dimension)
```

arrayName is the name of the array you want to determine bounds for, and *dimension* is the dimension you want to check (not required if you're evaluating a one-dimensional array). Both functions return integer values that can be assigned to variables or used in expressions.

The following program fragment uses the UBOUND and LBOUND functions to check a one-dimensional array named *players$*.

```
OPTION BASE 1
DIM players$(9)
PRINT "Upper bound is"; UBOUND(players$)
PRINT "Lower bound is"; LBOUND(players$)
```

When you run the program fragment, you receive the following output:

```
Upper bound is 9
Lower bound is 1
```

When you check the bounds of a two-dimensional array, you must supply a dimension number: 1 for the first dimension or 2 for the second dimension. The following program fragment uses UBOUND and LBOUND to check the bounds of both dimensions in a two-dimensional array named *janSales%*:

```
OPTION BASE 1
DIM janSales%(3, 4)
PRINT "Upper bound of first dimension is"; UBOUND(janSales%, 1)
PRINT "Lower bound of first dimension is"; LBOUND(janSales%, 1)
PRINT "Upper bound of second dimension is"; UBOUND(janSales%, 2)
PRINT "Lower bound of second dimension is"; LBOUND(janSales%, 2)
```

When you run the program fragment, you receive the following output:

```
Upper bound of first dimension is 3
Lower bound of first dimension is 1
Upper bound of second dimension is 4
Lower bound of second dimension is 1
```

Practice: Avoiding the subscript-out-of-range message

The BOUNDS.BAS program (Figure 8-17) demonstrates how to use UBOUND and LBOUND to check the bounds of an array in a DO loop and thus avoid out-of-range errors. The program dimensions a one-dimensional string array named *party$*, which holds party game ideas. BOUNDS.BAS then fills the array and prints it out.

NOTE: The DO loop in the BOUNDS program makes use of the EXIT DO statement, which exits the loop if the user enters QUIT *before all the elements of the array are filled. Use EXIT DO in your programs when you need to terminate a DO loop before the WHILE or UNTIL condition is met.*

Type in the BOUNDS.BAS program and run it.

```
' BOUNDS.BAS
' This program loads data into an array until QUIT is typed or
'   the array boundaries are exceeded.

OPTION BASE 1                ' set array base to 1
DIM party$(5)                ' dimension array with 5 elements

count% = 1                   ' initialize loop counter to 1

CLS                          ' display intro message and QUIT note
PRINT "Enter party game ideas; type QUIT to stop"
PRINT
                             ' loop while count% is within array bounds
DO WHILE (count% >= LBOUND(party$)) AND (count% <= UBOUND(party$))
    INPUT "Party game:  ", party$(count%)       ' read input into array
    IF (party$(count%) = "QUIT") THEN EXIT DO   ' if user types QUIT,
    count% = count% + 1                         '   then exit loop
LOOP

PRINT
PRINT "You entered the following games:"
PRINT

FOR i% = 1 TO count% - 1     ' print array contents
    PRINT party$(i%)         ' count% - 1 contains last value entered
NEXT i%
```

FIGURE 8-17.
BOUNDS.BAS: A program that uses LBOUND and UBOUND to avoid referencing an array element that is out of bounds.

You'll receive output similar to the following:

```
Enter party game ideas; type QUIT to stop

Party game:  Blindman's buff
Party game:  Bingo
Party game:  Pin the tail on the donkey
Party game:  Spin the bottle
Party game:  Moonlight bowling
```

(continued)

continued

```
You entered the following games:

Blindman's buff
Bingo
Pin the tail on the donkey
Spin the bottle
Moonlight bowling
```

SUMMARY

In this chapter you've worked with the important structures, functions, statements, and techniques QBasic provides for working with large amounts of data in a program:

- Using DATA, READ, and RESTORE to assign data stored in a program to variables
- Creating and using a one-dimensional array
- Searching for elements in an array
- Formatting tabular information with PRINT USING
- Creating and using a two-dimensional array
- Troubleshooting common array-related programming errors

In the next chapter you'll learn about the many QBasic functions and techniques especially designed for working with strings.

QUESTIONS AND EXERCISES

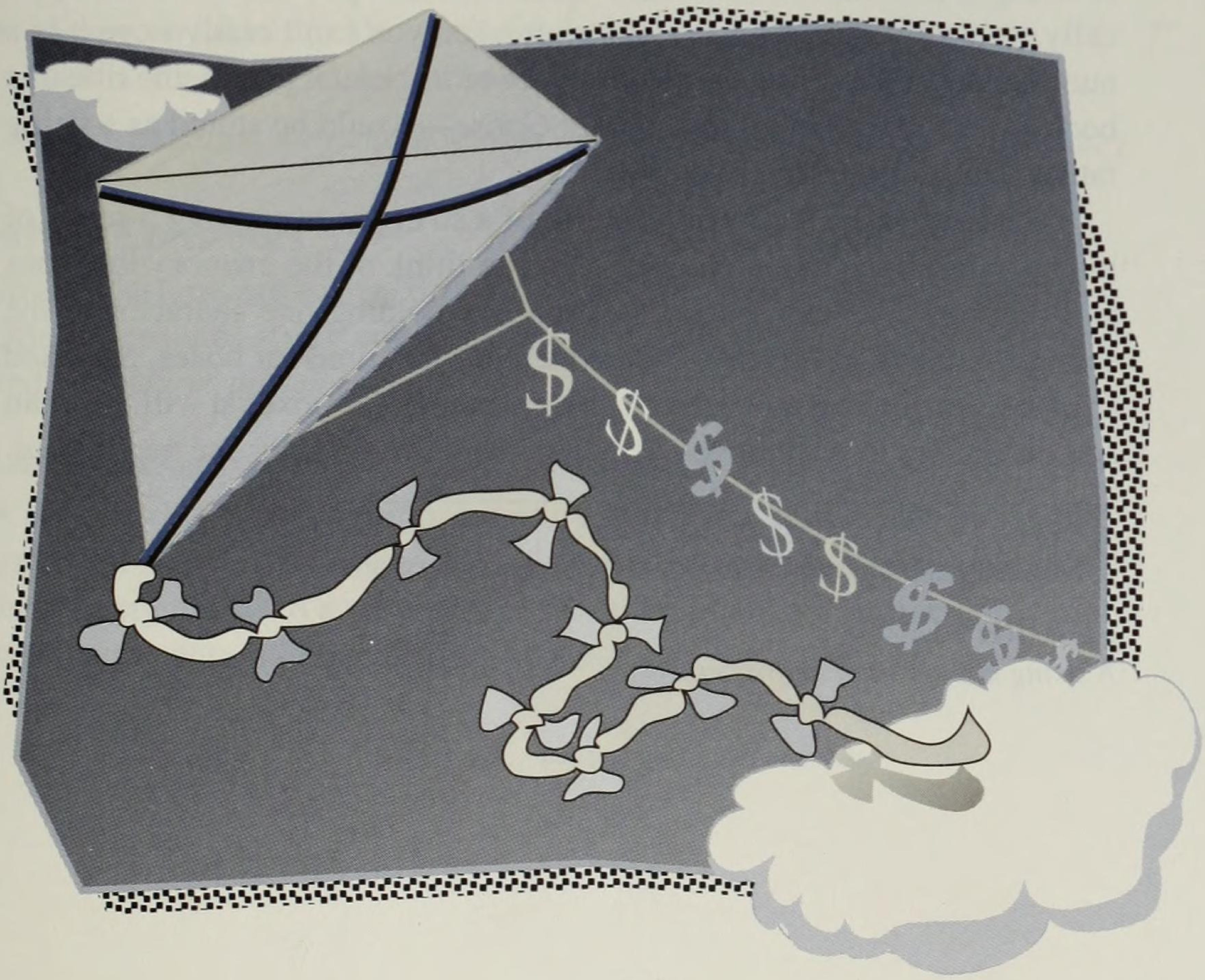
1. Which comes first in a program, a DATA statement or a READ statement?
2. What type of information is well suited to the DATA-READ-RESTORE approach to working with data?
3. Write a program that displays each of the values in the following DATA statement with a FOR loop:

   ```
   DATA Beaver, Wally, Lumpy, Whitey, Gus, Eddie, Larry
   ```
4. True or False: An array can store more than one type of data.

5. Write a statement that sets aside memory for an array of 100 single-precision floating-point values.
6. What is an end-of-data marker?
7. Write a program that declares, fills, and then prints a dynamic one-dimensional array containing the major characters of your favorite television show or movie.
8. What is an out-of-range error?
9. Write a program that uses a two-dimensional array to keep score for a nine-inning baseball game. Your program should initialize the array, prompt the user for the baseball teams' names and mascots, get the runs scored for each inning, determine the final score and winner of the game, and display the results.

CHAPTER 9

Working with Strings



Chapters 3 and 4 introduced the string data type and let you practice creating strings and displaying them on your screen. This chapter continues that discussion and introduces many of the functions available in QBasic for working with strings. We'll describe

- Assigning user input to a string
- Combining strings
- Selecting characters from a string
- Comparing strings
- Sorting strings

When combined with the skills you've developed in preceding chapters, these functions can help you perform a wide variety of tasks.

STRINGS: AN OVERVIEW

A string is a series of consecutive characters that you use as a unit. Typically, you store information as a string when you can't easily store it as a numeric data type. For example, because of its textual nature, the title of a book—let's say *The Original Mother Goose*—would be stored as a string rather than as a numeric data type.

You might find it helpful to think of a string as occupying a series of memory locations in a computer. You can think of the memory locations as little boxes set side by side—each box holding one character of the string, as shown in Figure 9-1. The memory locations, or boxes, are fixed in place, but you can move the characters among the boxes at will. You can add characters to or remove characters from the string as the need arises.

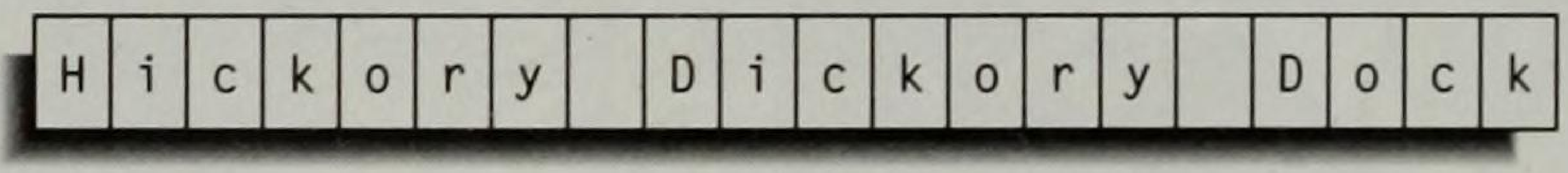


FIGURE 9-1.
A string is a series of characters.

You can use the following characters in a string:

- Uppercase letters of the alphabet (A though Z)
- Lowercase letters of the alphabet (a through z)
- Numerals (0 though 9)
- Punctuation symbols (. , ; : ‘ ’ ? !)
- Mathematical symbols (# % ( ) – + = \/< >)
- Miscellaneous symbols (~ @ $ ^ & * _ ¦ { } [ ])
- Characters in the IBM extended character set

TWO TYPES OF STRINGS

QBasic supports two types of strings for use in your programs:

- *String constants* are declared within your program. As the name suggests, the characters in a string constant never change.
- *String variables* are also declared within your program. The characters in a string variable can change at any time.

  Let’s examine why and how you use these types of strings in QBasic.

String Constants

String constants are strings that don’t change while a program is running. QBasic separates string constants into two groups: *literal string constants* and *symbolic string constants.*

Literal string constants

A literal string is a series of consecutive characters surrounded by double quotation marks. You generally assign literal strings to a variable or use them as arguments to a statement or function. For example, the following statements contain literal strings.

```
cheer$ = "Go Seahawks!"

PRINT "1313 Mockingbird Lane"

goal$ = CenterString$("Today I shall fly a kite")

birthDate$ = "11-19-63"
```

Literal strings are also known as *string values.*

Symbolic string constants

A symbolic string constant is a name assigned to a literal string:

```
CONST DI$ = "Her Royal Highness, Diana, Princess of Wales"
```

Constant

Literal string

After you've assigned a constant, you can use the constant anywhere you would otherwise have to use the literal string (much as you can call someone by a nickname as well as by a full name).

As you learned in Chapter 4, you assign a constant in a CONST statement, usually near the beginning of a program. In this book, constants are written in all-uppercase letters to distinguish them from variables:

```
CONST DI$ = "Her Royal Highness, Diana, Princess of Wales"

PRINT "Charles's wife is now called "; DI$
```

Once you declare a constant, you cannot change its value. For example, a program containing the following two statements would generate the error message *Duplicate definition* when you run the program because the value of a constant cannot change during the execution of a program:

```
CONST DI$ = "Her Royal Highness, Diana, Princess of Wales"
DI$ = "Lady Diana Spencer"
```

This is the difference between a constant and a variable: The value of a variable can change as a program runs; the value of a constant cannot.

Practice: Using string constants

Constants are particularly useful when you have repetitive patterns in a program. The PIG.BAS program (Figure 9-2) declares the constant *PIG$* and assigns to it the literal string *This little piggy*.

Type in the PIG.BAS program and run it.

```
' PIG.BAS
' This program demonstrates the use of string constants.

CLS

CONST PIG$ = "This little piggy "

PRINT PIG$; "went to market"
PRINT PIG$; "stayed home"
PRINT PIG$; "had roast beef"
PRINT PIG$; "had none"
PRINT PIG$; "cried 'Wee wee wee' all the way home"
```

FIGURE 9-2.
PIG.BAS: A simple demonstration of symbolic and literal string constants.

You'll receive the following output:

```
This little piggy went to market
This little piggy stayed home
This little piggy had roast beef
This little piggy had none
This little piggy cried 'Wee wee wee' all the way home
```

String Variables

Unlike string constants, string variables can change at any time during the execution of a program. QBasic supports two types of string variables: *variable-length strings* and *fixed-length strings*.

In this book, variable-length and fixed-length string variables begin with a lowercase letter to distinguish them from subprograms, functions, statements, and constants.

Variable-length strings

Variable-length strings let you obtain and store information from the user of a program. A variable-length string can contain from 0 through 32,767 characters (although the typical range is 0 through 80 because of the width of the screen), and its length can grow or shrink during the execution of a program. You can declare a variable-length string in three ways:

- By placing the string type-declaration character (*$*) at the end of the variable name. For example, the following statement gets a string from the user and assigns it to the string variable *firstName$*.

  ```
  INPUT "Enter your first name:  ", firstName$
  ```

  As you examine the examples in this book, you'll see many instances in which INPUT statements use variable-length strings to assign meaningful names to particular words and phrases. We also use these variable-length strings as arguments to statements and functions such as PRINT.

- By using the DEFSTR statement. For example, the following statement directs QBasic to regard as a variable-length string any variable that begins with the letter *S* and that does not have a type-declaration character (*%*, *&*, *!*, *#*, or *$*):

  ```
  DEFSTR S
  ```

- By using AS STRING with the DIM statement. For example, the following statement declares the variable *fullName* to be a variable-length string:

  ```
  DIM fullName AS STRING
  ```

  Note that *fullName* does not end with a *$*. Whenever you declare a string using the AS STRING keywords, you cannot use the string type-declaration character.

  As you learned in the previous chapter, you can also use the DIM statement to declare an *array* of variable-length strings. For example, the following statement declares an array of 10 variable-length strings (assuming that the statement OPTION BASE 1 appears first):

  ```
  DIM names$(10)
  ```

Practice: Using variable-length strings

The PHONE.BAS program (Figure 9-3) uses INPUT statements and a two-dimensional variable-length string array named *contacts$* to store a list of friends and their telephone numbers. (Because telephone numbers often contain characters that are not numerals, such as dashes and letters, it's a good idea to store them as strings.)

Type in the PHONE.BAS program and run it.

```
' PHONE.BAS
' This program uses a variable-length string array to record names and
'   telephone numbers.

OPTION BASE 1              ' set lower bound of array to 1

CLS

INPUT "How many names would you like to enter?  ", names%
PRINT

DIM contacts$(names%, 2)  ' declare array for names and phone numbers

FOR i% = 1 TO names%       ' read names into contacts$ array
    INPUT "Enter name:  ", contacts$(i%, 1)
    INPUT "Enter phone number:  ", contacts$(i%, 2)
    PRINT
NEXT i%

PRINT "You entered the following contact list:"
PRINT

FOR i% = 1 TO names%       ' print contents of array
    PRINT "Name:  "; contacts$(i%, 1), "Phone:  "; contacts$(i%, 2)
NEXT i%
```

FIGURE 9-3.
PHONE.BAS: A demonstration of variable-length strings.

You'll receive output similar to the following:

```
How many names would you like to enter?  3

Enter name:  Little Bo-Peep
Enter phone number:  555-LAMB

Enter name:  Little Jack Horner
Enter phone number:  555-PLUM

Enter name:  Little Boy Blue
Enter phone number:  555-HORN

You entered the following contact list:

Name:  Little Bo-Peep      Phone:  555-LAMB
Name:  Little Jack Horner  Phone:  555-PLUM
Name:  Little Boy Blue     Phone:  555-HORN
```

Note that you achieve the alignment of the phone-number column by using a comma in the PRINT statement to form two columns. In this case, advancing to the next print zone is enough to align the items. If the string values differed greatly in size, however, a different solution would be needed. Try entering some names of widely differing lengths to see why.

Fixed-length strings

Fixed-length strings let you declare a string variable of a certain length. Although the contents of a fixed-length string can change at any time, the length remains constant. In general, it is a good idea to use fixed-length strings whenever you are sure of the length of a string of characters or when you need to align groups of strings.

To declare a fixed-length string, you add a string variable name and a string length to the DIM statement, as follows:

```
DIM strName AS STRING * n
```

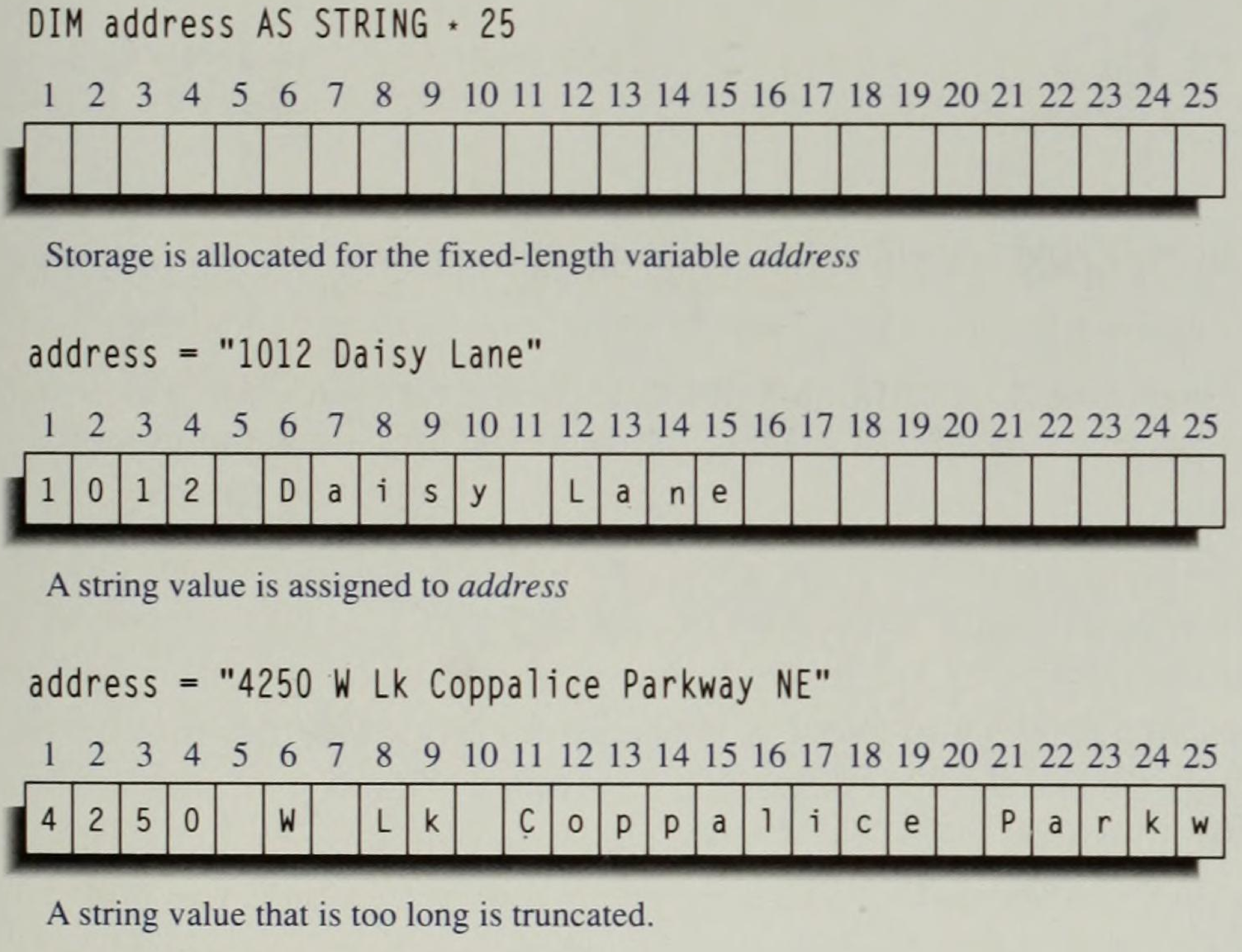
strName is the name of the fixed-length string, and *n* is the string length. For example, the following statement declares a fixed-length string named *address* that is 25 characters long:

```
DIM address AS STRING * 25
```

You can also dimension an *array* of fixed-length strings by including the number of array elements and a dimension. For example, the following statement declares a one-dimensional fixed-length string array named *fullNames*:

```
DIM fullNames(25) AS STRING * 25
```

Fixed-length strings are left-justified by default and are padded with blank spaces on the right side if the length of the string value assigned to the string is less that the total length of the string (Figure 9-4). Note that you cannot use the string type-declaration character (*$*) when you declare a fixed-length string.



```
DIM address AS STRING * 25
```

Storage is allocated for the fixed-length variable *address*

```
address = "1012 Daisy Lane"
```

A string value is assigned to *address*

```
address = "4250 W Lk Coppalice Parkway NE"
```

A string value that is too long is truncated.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
1 0 1 2 D a i s y L a n e
4 2 5 0 W L k C o p p a l i c e P a r k w

FIGURE 9-4.
Alignment of characters in a fixed-length string.

Practice: Fixed-length strings

The PHONEFIX.BAS program (Figure 9-5) revises the PHONE.BAS program to include two one-dimensional fixed-length string arrays to track the phone list. Two separate arrays are required because the string length in each array is different. If the string length were the same, you could use one two-dimensional fixed-length string array.

Type in the PHONEFIX.BAS program and run it.

```
' PHONEFIX.BAS
' This program uses two fixed-length string arrays to record names and
'   telephone numbers.

OPTION BASE 1                        ' set lower bound of array to 1

CLS
                                     ' get number of names
INPUT "How many names would you like to enter? ", names%
PRINT

DIM fullNames(names%) AS STRING * 18  ' 18 characters each for names
DIM phones(names%) AS STRING * 14     ' 14 characters each for phones

FOR i% = 1 TO names%                 ' read values into both arrays
    INPUT "Enter name:  ", fullNames(i%)
    INPUT "Enter phone number:  ", phones(i%)
    PRINT
NEXT i%

PRINT "Name                Phone"
PRINT "-----------------------------------"

FOR i% = 1 TO names%                 ' print contents of both arrays
    PRINT fullNames(i%); "  "; phones(i%)
NEXT i%
```

FIGURE 9-5.
PHONEFIX.BAS: A demonstration of fixed-length strings.

You'll receive output similar to the following:

```
How many names would you like to enter? 3

Enter name:  Little Bo-Peep
Enter phone number:  555-LAMB

Enter name:  Little Jack Horner
Enter phone number:  555-PLUM

Enter name:  Little Boy Blue
Enter phone number:  555-HORN

Name                Phone
-------------------------------------
Little Bo-Peep      555-LAMB
Little Jack Horner  555-PLUM
Little Boy Blue     555-HORN
```

Note that the alignment of the phone-number column is now predictable. If one of the names or phone numbers were longer than the allotted number of characters, it would be truncated.

Combining Strings

One of the simplest things you can do with strings is to combine them to form longer strings. This process is called *concatenation.* You concatenate strings primarily to prepare text for output on the screen or printer or to prepare the strings for storage in a data structure or a file. A concatenated string generally should contain 80 characters or fewer. This limitation prevents the string from wrapping (spilling over onto the next line).

You can concatenate both string constants and string variables in any combination and then assign the result to a string variable or supply the result as an argument to a statement or function that expects string values (such as PRINT).

For example, the following statement uses the concatenation operator (+) to combine the literal strings *Programming*, *is*, and *fun* and assigns the result to the string variable *phrase$*:

```
phrase$ = "Programming" + "is" + "fun"
```

The concatenation operator combines the three literal strings to form one string. The assignment operator (=) then assigns the result to the string variable *phrase$*. Note that there are no spaces in the literal strings in this statement. (The spaces surrounding the operators don't count.) If you were to print out the value of *phrase$* by means of the PRINT statement, the following output would appear:

```
Programmingisfun
```

The concatenation operator combines the strings exactly as they are, without adding spaces. To include spaces, you must add spaces to the literal strings themselves; that is, a space needs to appear *within* the quotation marks. Each of the following statements would accomplish this:

- `PRINT "Programming" + " is" + " fun"`
- `PRINT "Programming " + "is " + "fun"`
- `PRINT "Programming" + " " + "is" + " " + "fun"`

You can also supply the the result of a concatenation as an argument to some QBasic statements directly—without assigning the result to an intermediate variable such as *phrase$*:

```
PRINT "Programming " + "is " + "fun"
```

The result of the preceding statement is the same as the result of

```
phrase$ = "Programming " + "is " + "fun"
PRINT phrase$
```

Practice: Concatenating strings

The NEWS.BAS program (Figure 9-6) demonstrates a number of the options available to you through string concatenation.

Type in the NEWS.BAS program and run it.

```
' NEWS.BAS
' This program demonstrates string concatenation.

CONST STRUCTURE$ = "Bridge"

DIM action AS STRING * 10
action = "is falling"

direction$ = "down"

CLS

INPUT "Please enter the name of a city:  ", cityName$

PRINT
PRINT "News Flash:  ";
PRINT cityName$ + " " + STRUCTURE$ + " " + action + " " + direction$ + "!"
```

FIGURE 9-6.
NEWS.BAS: A program demonstrating string concatenation.

If you respond to the city prompt with *London*, you'll receive the following output:

```
Please enter the name of a city:  London
News Flash:  London Bridge is falling down!
```

PUTTING STRING FUNCTIONS TO WORK

So far you've declared string constants and variables, combined strings in a process called concatenation, and used strings as arguments in INPUT and PRINT statements. In this section you'll learn about QBasic functions that are specifically designed to manipulate and return values from literal strings and string variables. You'll learn how to

- Change the case of a string
- Determine the length of a string
- Take strings apart

Changing the Case of a String

It's easy to change a string's letters to all uppercase or all lowercase. Simply use the UCASE$ or LCASE$ function. These functions are handy if you need to ensure that all data in your program is in the same format and you're not sure whether the user will enter uppercase or lowercase text. When you learn at the end of this chapter how strings are compared, you'll understand the importance of this information.

Using the UCASE$ function

To change a string's letters to all uppercase, use the UCASE$ function, as follows:

```
UCASE$(stringexpression)
```

stringexpression is any variety of string. The value returned by UCASE$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values. UCASE$ affects only the lowercase letters of *stringexpression.*

Practice: Using the UCASE$ function

The UPPER.BAS program (Figure 9-7) demonstrates how the UCASE$ function works. UPPER.BAS declares one string constant and two string variables and then uses the UCASE$ function to display them in uppercase. Note that the UCASE$ function affects only the output of the PRINT statement—it doesn't change the contents of the *WRITER$* constant or the *address$* variable. (It's a good thing too, because—as you've learned—assigning a new value to a constant would be an error.) When the strings are displayed again at the end of the program, only the *borough$* variable retains uppercase letters, by virtue of its original string assignment.

Type in the UPPER.BAS program and run it.

```
' UPPER.BAS
' This program demonstrates the UCASE$ function.

CONST WRITER$ = "Sir Arthur Conan Doyle"
address$ = "1326 Serpentine Avenue"
borough$ = UCASE$("St. John's Wood")

CLS

PRINT UCASE$(WRITER$)
PRINT UCASE$(address$) + ", " + borough$
PRINT
PRINT WRITER$
PRINT address$ + ", " + borough$
```

FIGURE 9-7.
UPPER.BAS: A demonstration of the UCASE$ function.

You'll receive the following output:

```
SIR ARTHUR CONAN DOYLE
1326 SERPENTINE AVENUE, ST. JOHN'S WOOD

Sir Arthur Conan Doyle
1326 Serpentine Avenue, ST. JOHN'S WOOD
```

Using the LCASE$ function

To change a string's letters to all lowercase, use the LCASE$ function, as follows:

`LCASE$(stringexpression)`

stringexpression is any variety of string. The value returned by LCASE$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values. LCASE$ affects only the uppercase letters of *stringexpression.*

Practice: Using the LCASE$ function

The LOWER.BAS program (Figure 9-8) demonstrates how the LCASE$ function works. LOWER.BAS converts the uppercase letters in a string to lowercase and assigns the resulting string to a variable and the PRINT statement.

Type in the LOWER.BAS program and run it.

```
' LOWER.BAS
' This program demonstrates the UCASE$ and LCASE$ functions.

CLS

character$ = "Sherlock Holmes"
PRINT character$

character$ = UCASE$(character$)
PRINT character$

character$ = LCASE$(character$)
PRINT character$
```

FIGURE 9-8.
LOWER.BAS: A program demonstrating the LCASE$ function.

You'll receive the following output:

```
Sherlock Holmes
SHERLOCK HOLMES
sherlock holmes
```

Determining the Length of a String

Often you'll want to know how many characters are in a string. This knowledge can be particularly handy with variable-length strings, especially those entered from the keyboard.

To determine how many characters (including spaces) are in a string, use the LEN function, as follows:

`LEN(`*`stringexpression`*`)`

stringexpression once again is any kind of string. The value returned by LEN can be assigned to an integer variable or supplied as an argument to a statement or function that accepts integer values.

The following routine shows how the LEN function determines the number of characters in a string and assigns the number to an integer variable:

```
fullName$ = "Old Mother Hubbard"
nameLength% = LEN(fullName$)
PRINT fullName$; " is"; nameLength%; "characters long"
```

When you execute this routine, you receive the following output:

```
Old Mother Hubbard is 18 characters long
```

Practice: Using the LEN function

The MENU.BAS program (Figure 9-9) shows how the value returned by the LEN function can be used as an argument to a PRINT statement. MENU.BAS declares three variable-length strings that contain potential meal selections for the user. These strings are displayed with PRINT statements. An INPUT statement prompts the user for a meal selection. The user's selection is assigned to the *choice$* variable and is converted to uppercase with the UCASE$ function. The SELECT CASE statement then compares the *choice$* variable with the three meal variables. If a match occurs, the meal choice and its length (returned by the LEN function) are printed. If no match occurs, the CASE ELSE statement prints the message *I don't recognize that meal!*

Type in the MENU.BAS program and run it.

```
' MENU.BAS
' This program demonstrates the LEN function.

missMuffet$ = "Curds and Whey"         ' declare meal strings
simpleSimon$ = "Pie du Jour"
petersPlate$ = "Pumpkin Surprise"
```

FIGURE 9-9. *(continued)*
MENU.BAS: A program demonstrating the LEN function.

FIGURE 9-9. *continued*

```
CLS

PRINT "Which meal would you enjoy?"    ' display meal prompt
PRINT

PRINT missMuffet$                      ' display meal choices
PRINT simpleSimon$
PRINT petersPlate$
PRINT

INPUT "Selection:  ", choice$          ' get meal choice from user
choice$ = UCASE$(choice$)              ' convert string to uppercase
PRINT

SELECT CASE choice$                    ' find variable that matches
    CASE IS = UCASE$(missMuffet$)      '   meal entered and print length
        PRINT missMuffet$; " is"; LEN(missMuffet$); "characters long"
    CASE IS = UCASE$(simpleSimon$)
        PRINT simpleSimon$; " is"; LEN(simpleSimon$); "characters long"
    CASE IS = UCASE$(petersPlate$)
        PRINT petersPlate$; " is"; LEN(petersPlate$); "characters long"
    CASE ELSE                          ' print message if no match found
        PRINT "I don't recognize that meal!"
END SELECT
```

You'll receive output similar to the following:

```
Which meal would you enjoy?

Curds and Whey
Pie du Jour
Pumpkin Surprise

Selection:  Curds and Whey

Curds and Whey is 14 characters long
```

Taking Strings Apart

Earlier in this chapter you learned that QBasic lets you combine strings through concatenation. But sometimes you'll want to take a string apart—to get only a person's last name from the full name, for example. QBasic provides six functions that allow you to work with parts of strings. The following sections describe how to use these functions to

- Get the right end of a string (RIGHT$)
- Get the left end of a string (LEFT$)
- Get the middle of a string (MID$)
- Trim the right end of a string (RTRIM$)
- Trim the left end of a string (LTRIM$)
- Find a string within a string (INSTR)

You'll also learn about statements that let you

- Get an entire line of input (LINE INPUT$)
- Print repeated characters (SPACE$, STRING$)

Getting the ends of a string

The RIGHT$ and LEFT$ functions let you retrieve one or more characters starting from one end of a string. This is useful when you want to display only part of a string or when you want to remove part of a string.

The syntax for the RIGHT$ function is as follows:

```
RIGHT$(stringexpression, n)
```

The syntax for the LEFT$ function is as follows:

```
LEFT$(stringexpression, n)
```

stringexpression is any variety of string, and *n* is an integer value (ranging from 0 through the length of the string) indicating the number of characters to be returned by RIGHT$ or LEFT$. The value returned can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

Practice: Using the RIGHT$ function

The GETRIGHT.BAS program (Figure 9-10) uses the RIGHT$ function to retrieve characters from a variable named *alphabet$*, which contains the 26 letters of the alphabet. GETRIGHT.BAS extracts the requested number of characters and displays them with a character count.

Type in the GETRIGHT.BAS program and run it.

```
' GETRIGHT.BAS
' This program demonstrates the RIGHT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from right to left) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

    ' get from user number of rightmost characters to be displayed
DO  ' loop until number is in proper range (1 through 26)
    INPUT "    Number (1-26):  ", rightNum%
LOOP WHILE (rightNum% < 1) OR (rightNum% > 26)

PRINT
rightChar$ = RIGHT$(alphabet$, rightNum%)  ' display characters
PRINT "You specified"; LEN(rightChar$); "characters:  "; rightChar$
```

FIGURE 9-10.
GETRIGHT.BAS: A program demonstrating the RIGHT$ function.

You'll receive output similar to the following:

```
How many characters (from right to left) in the following
string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26):  14

You specified 14 characters:  MNOPQRSTUVWXYZ
```

Practice: Using the LEFT$ function

The GETLEFT.BAS program (Figure 9-11) revises the GETRIGHT.BAS program to extract characters from the left side of a string with the LEFT$ function. Note that the variable names have changed slightly (*rightNum%* becomes *leftNum%*, and *rightChar$* becomes *leftChar$*) and that the function RIGHT$ has been changed to LEFT$. Apart from these changes (and a few changes to the prompt and program comments), GETLEFT.BAS is identical to GETRIGHT.BAS. Because the operations of the RIGHT$ and LEFT$ functions are so similar, it's quite easy to change a program so that it modifies a string from the opposite end.

Modify the GETRIGHT.BAS program to match the GETLEFT.BAS program and run it.

```
' GETLEFT.BAS
' This program demonstrates the LEFT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

    ' get from user number of leftmost characters to be displayed
DO  ' loop until number is in proper range (1 through 26)
    INPUT "    Number (1-26):  ", leftNum%
LOOP WHILE (leftNum% < 1) OR (leftNum% > 26)

PRINT
leftChar$ = LEFT$(alphabet$, leftNum%)     ' display characters
PRINT "You specified"; LEN(leftChar$); "characters:  "; leftChar$
```

FIGURE 9-11.
GETLEFT.BAS: A program demonstrating the LEFT$ function.

You'll receive output similar to the following:

```
How many characters (from left to right) in the following
string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26):  14

You specified 14 characters:  ABCDEFGHIJKLMN
```

Getting the middle of a string

The MID$ function lets you retrieve one or more characters from anywhere within a string—from the left, from the middle, or (with some help from the LEN function) from the right. Its versatility makes the MID$ function one of the most useful string functions. And, as we'll see later, it provides the processing power to solve many string-related problems.

The syntax for the MID$ function is as follows:

```
MID$(stringexpression, start, length)
```

stringexpression is any kind of string, *start* is an integer value between 1 and the length of the string (indicating the first character to be returned), and *length* is an integer value indicating the number of characters to be returned. The value returned by MID$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values. Figure 9-12 shows the elements of the MID$ function syntax.

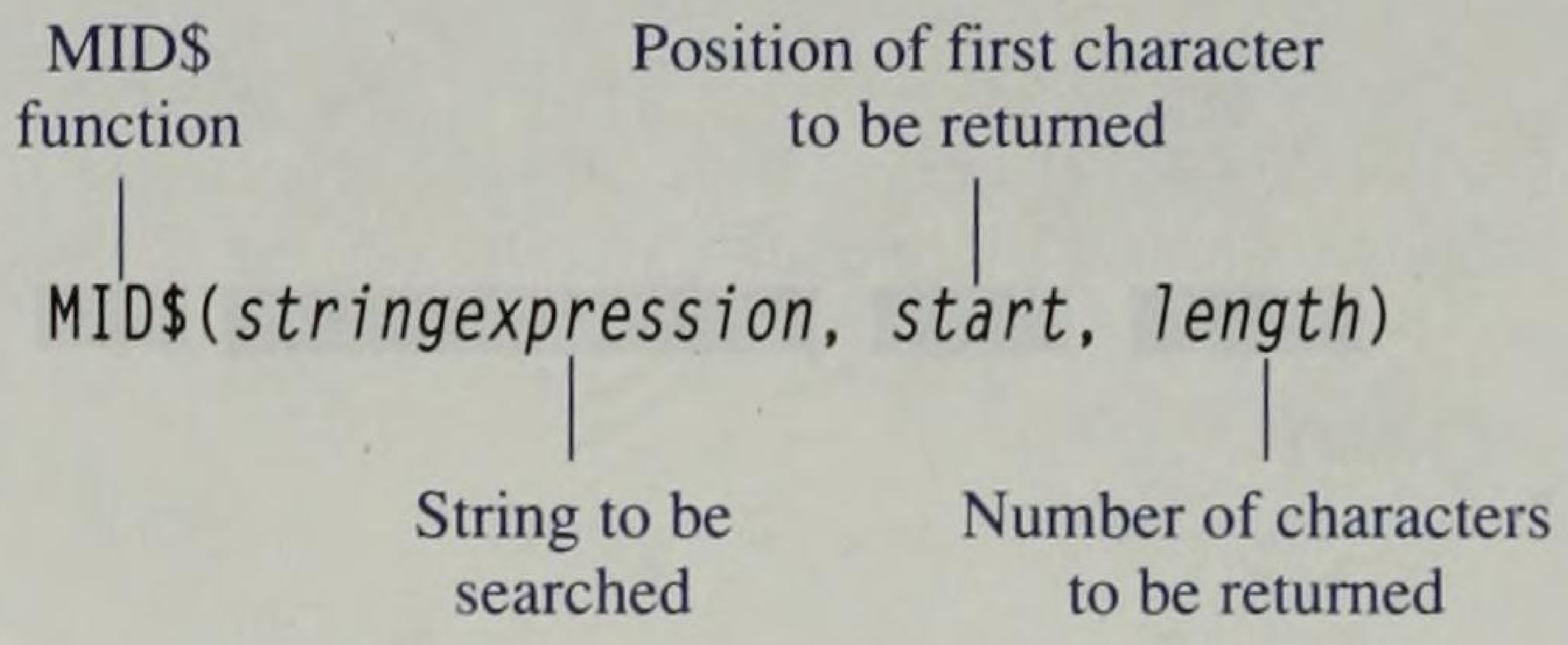

MID$ function
Position of first character to be returned
MID$(stringexpression, start, length)
String to be searched
Number of characters to be returned

FIGURE 9-12.
The components of the MID$ function.

The following statements show some valid uses of the MID$ function. Notice the powerful possibilities that arise when you use the value returned by a function as an argument to MID$ or when you assign the value returned by MID$ to another statement or function.

```
middleName$ = MID$("Queen Victoria Belfield", 7, 8)
```

Result: *middleName$* contains `Victoria`

```
address$ = "1521 Plumtree Lane #25-K"
streetNameStart% = 6
length% = 13
PRINT UCASE$(MID$(address$, streetNameStart%, length%))
```

Result: `PLUMTREE LANE`

```
inString$ = "Making it all make sense"
rightmostWord$ = MID$(inString$, LEN(inString$) - 4, 5)
```

Result: *rightmostWord$* contains `sense`

```
PRINT "The current year is "; MID$(DATE$, 7, 4)
```

Result: `The current year is 1991`

Practice: Using the MID$ function

The GETMID.BAS program (Figure 9-13 on the following page) shows how to retrieve characters from the middle of a string with the MID$ function. GETMID modifies the GETRIGHT and GETLEFT programs to include a starting point along with the number of characters in the *alphabet$* string to be displayed. GETMID uses two DO loops to get integer values in the proper range and then uses the MID$ function to assign the selected characters to the *midChar$* variable. The results of the selection are printed with the following IF statement, which appears near the end of the program:

```
IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed:  "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed:  "; midChar$
END IF
```

The IF statement compares *numToDisplay%*—the variable containing the number of characters the user requested to be displayed—to the number of characters in *midChar$* (returned by the LEN function). If the two values are equal, the value of *numToDisplay%* is printed along with the contents of *midChar$*. If the two values are not equal, the LEN function is used to determine the actual number of characters, and this value is displayed along with the *midChar$* string. GETMID contains this additional message to notify the user that the display length entered exceeded the number of characters remaining in the string.

Modify the GETLEFT.BAS program to match the GETMID.BAS program and run it.

```
' GETMID.BAS
' This program demonstrates the MID$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                           ' display test string
PRINT

    ' get from user number of characters to be displayed
DO  ' loop until number is in proper range (1 through 26)
    INPUT "    Number (1-26):  ", numToDisplay%
LOOP WHILE (numToDisplay% < 1) OR (numToDisplay% > 26)
PRINT                                     ' get starting number...
PRINT "What character would you like to start with?"
PRINT

DO                                        '   in proper range
    INPUT "    Starting number (1-26):  ", start%
LOOP WHILE (start% < 1) OR (start% > 26)
```

FIGURE 9-13. *(continued)*

GETMID.BAS: A program demonstrating the MID$ function.

FIGURE 9-13. *continued*

```
PRINT                                  ' get characters
midChar$ = MID$(alphabet$, start%, numToDisplay%)

    ' compare requested characters with actual characters retrieved
    '   and print an appropriate message with string
IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed:  "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed:  "; midChar$
END IF
```

You'll receive output similar to the following:

```
How many characters (from left to right) in the following
string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26):  14

What character would you like to start with?

    Starting number (1-26):  4

 14 characters displayed:  DEFGHIJKLMNOPQ
```

If you specify a number outside the range permitted by GETMID, you receive output similar to the following:

```
How many characters (from left to right) in the following
string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26):  0
    Number (1-26):  30
    Number (1-26):  15
```

(continued)

continued

```
What character would you like to start with?

    Starting number (1-26):  w

Redo from start
    Starting number (1-26):  20

 15 characters requested, 7 displayed:  TUVWXYZ
```

If you type in a non-numeric value (such as a letter) at one of the prompts, QBasic prints the message *Redo from start* and redisplays the prompt. Handling this type of response from the user is important in developing "break-proof" programs that other people will use.

Trimming a string

When you want to align strings or eliminate wasted space at the ends of strings, you can use the QBasic trimming functions, LTRIM$ and RTRIM$.

The LTRIM$ function lets you trim the left end of a string; that is, LTRIM$ returns a string with no leading spaces or tabs. The RTRIM$ function performs a similar task on the right end of a string.

The syntax for the LTRIM$ function is as follows:

```
LTRIM$(stringexpression)
```

The syntax for the RTRIM$ function is as follows:

```
RTRIM$(stringexpression)
```

stringexpression can be any variety of string. The value returned by LTRIM$ or RTRIM$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

The following routine uses the LTRIM$ function to remove the three leading spaces from the string variable *englishPhrase$* before printing:

```
englishPhrase$ = "   Good evening"
englishPhrase$ = LTRIM$(englishPhrase$)

PRINT englishPhrase$; "!"
```

This routine results in the following output:

```
Good evening!
```

The following routine uses the RTRIM$ function to remove the three final spaces from the string variable *germanPhrase$* before printing:

```
germanPhrase$ = "Guten Abend   "
germanPhrase$ = RTRIM$(germanPhrase$)

PRINT germanPhrase$; "!"
```

This routine results in the following output:

```
Guten Abend!
```

Practice: Using LTRIM$ and RTRIM$

The GREETING.BAS program (Figure 9-14) shows how to use the LTRIM$ and RTRIM$ functions together to trim blank spaces from both ends of a fixed-length string.

GREETING declares two fixed-length string variables named *firstName* and *lastName* and uses the INPUT statements to obtain string values for these variables. The GREETING program then displays two messages for the user of the program. The first message is not trimmed; the second is trimmed by both the LTRIM$ and RTRIM$ functions. Note that the fixed-length strings themselves are not modified—you cannot change the length of a fixed-length string.

Note how a string argument is passed from RTRIM$ to LTRIM$ to PRINT in the following statement from GREETING:

```
PRINT "I think "; LTRIM$(RTRIM$(firstName)); " ";
```

The RTRIM$ function processes the variable *firstName* and passes the resulting string value to the LTRIM$ function. LTRIM$ then processes the string and passes the string value to the PRINT statement. The PRINT statement then combines the string value with two other string values and displays them on your screen.

Type in the GREETING.BAS program and run it.

```
' GREETING.BAS
' This program demonstrates the LTRIM$ and RTRIM$ functions.

DIM firstName AS STRING * 12  ' declare fixed-length strings
DIM lastName AS STRING * 15

CLS
                              ' get first and last names
INPUT "Enter your first name:  ", firstName
INPUT "Enter your last name:  ", lastName

PRINT                         ' print greeting without trimming
PRINT "Nice to meet you, "; firstName; " "; lastName; "!"
                              ' print compliment with trimming
PRINT "I think "; LTRIM$(RTRIM$(firstName)); " ";
PRINT LTRIM$(RTRIM$(lastName)); " has a nice ring to it."
```

FIGURE 9-14.
GREETING.BAS: A program demonstrating the LTRIM$ and RTRIM$ functions.

You'll receive output similar to the following:

```
Enter your first name:  Jack
Enter your last name:  Sprat

Nice to meet you, Jack         Sprat          !
I think Jack Sprat has a nice ring to it.
```

Getting an entire line of input from the user

Throughout this chapter we've used the INPUT statement to get input from the user. The INPUT statement is quite versatile—it can assign input to one or more variables of different types and supply an optional prompt to spell out exactly what the user should enter. One thing we haven't discussed, however, is how the INPUT statement processes a comma in the input line. Consider the following statement, which prompts the user to enter a name and address:

```
INPUT "Enter name and address:  ", mailingAddress$
```

If the user responds to the prompt with a string containing commas, the error message *Redo from start* appears, as shown in the following dialog:

```
Enter name and address:  Jon Victor, 1118 Skyridge, Lacey, WA, 98503

Redo from start
Enter name and address:
```

As we noted in Chapter 4, the INPUT statement has a special use for the comma character in the input line: The comma separates the values assigned to variables. But what happens when unexpected commas appear in the input—as shown above? In this case you could solve the problem by assigning parts of the input string to different variables. For example, the following INPUT statement assigns the string value entered by the user to five string variables:

```
INPUT "Enter name and address:  ", name$, address$, city$, state$, zip$
```

But there are times when it would be a lot simpler to have only one variable name associated with a line of input. QBasic provides a solution to this problem with the LINE INPUT statement. The LINE INPUT statement gets an entire line of text from the keyboard and assigns it to a string variable, even if commas are present.

The syntax for the LINE INPUT statement is as follows:

```
LINE INPUT [;] ["promptstring";] stringvariable
```

promptstring is a literal string that prompts the user for input, and *stringvariable* is any string variable.

- Placing a semicolon immediately after LINE INPUT keeps the cursor on the same line after the user presses Enter.
- If *promptstring* is included in the statement, a semicolon is required to separate *promptstring* from *stringvariable.*
- Unlike the INPUT statement, the LINE INPUT statement prints no question mark unless the question mark is included in *promptstring*.

The following routine demonstrates the usefulness of LINE INPUT for long lines of input containing the comma character:

```
LINE INPUT "Enter name and address:  "; mailingAddress$
PRINT mailingAddress$
```

When you execute the routine and enter the information we tried to enter earlier, you'll receive the following output:

```
Enter name and address:  Jon Victor, 1118 Skyridge, Lacey, WA, 98503
Jon Victor, 1118 Skyridge, Lacey, WA, 98503
```

You'll see the LINE INPUT statement from time to time in later chapters.

Printing repeated characters

QBasic provides two useful functions that generate strings of repeated characters: the SPACE$ function, which returns a string of spaces, and the STRING$ function, which returns a string of characters. Both functions provide fast ways to build strings for use in formatting and aligning your program's output.

The syntax of the SPACE$ function is as follows:

```
SPACE$(n)
```

n is an integer value specifying the number of spaces the string will contain. The value returned by SPACE$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

The most common use of the SPACE$ function is in formatting output, as shown in the following routine:

```
blank$ = SPACE$(15)
PRINT blank$; "Big sale on bunnies today!"
```

You can use SPACE$ whenever you consistently indent text a set number of spaces.

The syntax of the STRING$ function is as follows:

```
STRING$(m, stringexpression)
```

m is an integer value specifying the length of the string to be returned, and *stringexpression* is the character to be repeated. The value returned by STRING$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

Practice: Using SPACE$ and STRING$

The most common use of the STRING$ function is for headings used in program output. The HEADER.BAS program (Figure 9-15) uses STRING$ to display a header message in the middle of the screen. Note that using a constant to specify the number of repeated characters makes it easy to modify the program later and that the variables created by SPACE$ and STRING make excellent candidates for concatenation.

Type in the HEADER.BAS program and run it.

```
' HEADER.BAS
' This program demonstrates the SPACE$ and STRING$ functions.

CONST LENGTH% = 16

fileName$ = "SWEETPEA.DOC"
blank$ = SPACE$(LENGTH%)
asterisk$ = STRING$(LENGTH%, "*")
banner$ = blank$ + asterisk$ + " " + fileName$ + " " + asterisk$

CLS

PRINT banner$
```

FIGURE 9-15.
HEADER.BAS: A program demonstrating the SPACE$ and STRING$ functions.

You'll see the filename *SWEETPEA.DOC* in the middle of the top line of the screen, surrounded by equal numbers of asterisks and blank spaces.

Finding a string within a string

We've used the RIGHT$, LEFT$, and MID$ functions to return characters from the right, left, and middle portions of a string. These functions are quite effective at extracting characters from a set place in a string, but they are less effective at searching for and extracting a specific pattern from a string. QBasic fills this gap with the INSTR function, which searches for one string within another string. The INSTR function joins RIGHT$, LEFT$, MID$, and the support functions we've discussed in this section (UCASE$, LCASE$, LEN$, RTRIM$, LTRIM$, SPACE$, and STRING$) to round out a complete collection of tools for working with strings.

The syntax for the INSTR function is as follows:

```
INSTR([start, ]basestring, searchstring)
```

start is an optional integer value specifying the character at which the search should begin, *basestring* is the string being searched, and *searchstring* is the string being sought. The value returned by INSTR can be assigned to an integer variable or supplied as an argument to a statement or function that accepts integer values. The following table lists the values that the INSTR function can return:

Condition	*Integer value returned*
searchstring found in *basestring*	Position in *basestring* at which match is found
searchstring not found in *basestring*	0
start is greater than length of *basestring*	0
basestring contains no characters	0
searchstring contains no characters	*start* (if given); otherwise, 1

Practice: Using the INSTR function

The HUNT.BAS program (Figure 9-16) uses the INSTR function to search for the string *iddle* in the string *High, diddle, diddle, the cat and the fiddle.*

Type in the HUNT.BAS program and run it.

```
' HUNT.BAS
' This program demonstrates the INSTR function.

CLS

search$ = "iddle"
base$ = "High, diddle, diddle, the cat and the fiddle"
strLocation% = INSTR(1, base$, search$)

IF (strLocation% <> 0) THEN
    PRINT search$; " first appears starting at character"; strLocation%
ELSE
    PRINT search$; " not found"
END IF
```

FIGURE 9-16.
HUNT.BAS: A program demonstrating the INSTR function.

When you run the program, you receive the following output:

```
iddle first appears starting at character 8
```

Although the pattern *iddle* appears three times in *base$*, the INSTR function returns the location of only the first occurrence. However, you can use INSTR within a loop to find multiple occurrences of a pattern.

It's a good idea to check the value returned by INSTR whenever you use the INSTR function. This will allow your program to take appropriate action if the search string is not found, or if either the search string or the base string is empty.

For example, in the HUNT program, if *iddle* were not found, the INSTR function would assign the value 0 to the *strLocation%* variable, and the IF statement would display the following message to indicate that *iddle* did not exist in *base$*:

```
iddle not found
```

Practice: Finding multiple occurrences of a pattern

The REPEAT.BAS program (Figure 9-17) uses the INSTR function to find multiple occurrences of a pattern in a series of lines entered from the keyboard.

The heart of REPEAT is found in the *Repeat%* function. Each time INSTR finds the search string (*search$*) in the base string (*base$*), the location of the search string is assigned to the *currentChar%* variable. Then the *num%* variable is incremented and the *currentChar%* variable is moved ahead to the position just after the string match in the base string (the new starting place for the next search). This process continues until the end of the base string is reached or the search string is not found in the remainder of the base string. The *Repeat%* function then returns the total number of matches in the line to the *lineRepeats%* variable in the main program.

Type in the REPEAT.BAS program and run it. Note that Figure 9-17 shows how REPEAT.BAS looks when it is printed out—not how it looks inside QBasic. Be sure to use the New SUB command on the Edit menu to enter the *GetText* subprogram and the New FUNCTION command on the Edit menu to enter the *Repeat%* function. You can review this process in Chapter 7.

```
' REPEAT.BAS
' This program prompts the user for a set number of lines and a search
'   string and then prints the lines and the number of matches found.

' set maximum number of lines that can be entered and declare string
'   array to hold lines

CONST MAXLINES% = 10
DIM inputLines$(MAXLINES%)

' declare GetText subprogram and Repeat% function

DECLARE SUB GetText (inputLines$(), numOfLines%)
DECLARE FUNCTION Repeat% (search$, base$)

CLS
```

FIGURE 9-17. *(continued)*
REPEAT.BAS: A program that scans lines for a pattern and prints the number of matches.

FIGURE 9-17. *continued*

```
' call GetText subprogram to get input from user; at return, the
'   numOfLines% variable will contain number of lines received

GetText inputLines$(), numOfLines%

' get pattern to be searched for from user

PRINT
INPUT "Enter the string to be searched for:  ", pattern$
PRINT

' call Repeat% function to determine the number of matches per line;
'   totalRepeats% variable tracks number of total matches

FOR i% = 1 TO numOfLines%
    lineRepeats% = Repeat%(pattern$, inputLines$(i%))
    totalRepeats% = totalRepeats% + lineRepeats%
NEXT i%

' display lines entered by the user...

PRINT "You entered the following lines:"
PRINT
FOR i% = 1 TO numOfLines%
    PRINT inputLines$(i%)
NEXT i%

' and the total number of matches

PRINT
PRINT "The pattern '"; pattern$; "' appears"; totalRepeats%; "times"

END

SUB GetText (strArray$(), count%)

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard. The number of lines that can be
'   entered is determined by the global constant MAXLINES%.
```

(continued)

FIGURE 9-17. *continued*

```
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

PRINT "Enter up to"; MAXLINES%; "lines of text; to end, ";
PRINT "press Enter on a new line."
PRINT
count% = 0

DO
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN    ' if line is not blank, copy it
        count% = count% + 1    '   to the strArray$ array
        strArray$(count%) = inLine$
    END IF
' loop until count% = MAXLINES% or an empty line is received
LOOP WHILE (count% < MAXLINES%) AND (inLine$ <> "")

END SUB

FUNCTION Repeat% (search$, base$)

' The Repeat% function returns the number of times a search string
'   is found in a base string.

searchLength% = LEN(search$)  ' determine length of search string
baseLength% = LEN(base$)      ' determine length of base string
currentChar% = 1              ' character offset in base string
num% = 0                      ' running total of matches found in base

' loop until entire string is processed or INSTR returns a 0

WHILE (currentChar% <= baseLength%) AND (currentChar% <> 0)
    currentChar% = INSTR(currentChar%, base$, search$)
    IF (currentChar% <> 0) THEN  ' if not 0, a match was found
        num% = num% + 1          ' increment number of matches
        ' new offset equals current offset plus search-string length
        currentChar% = currentChar% + searchLength%
    END IF
WEND
```

(continued)

FIGURE 9-17. *continued*

```
Repeat% = num%                    ' return total number of matches

END FUNCTION
```

You'll receive output similar to the following:

```
Enter up to 10 lines of text; to end, press Enter on a new line.

-> Ten lords a-leaping,
-> Nine ladies dancing,
-> Eight maids a-milking,
-> Seven swans a-swimming,
-> Six geese a-laying,
-> Five gold rings,
-> Four calling birds,
-> Three French hens,
-> Two turtledoves, and
-> A partridge in a pear tree.

Enter the string to be searched for:  ing

You entered the following lines:

Ten lords a-leaping,
Nine ladies dancing,
Eight maids a-milking,
Seven swans a-swimming,
Six geese a-laying,
Five gold rings,
Four calling birds,
Three French hens,
Two turtledoves, and
A partridge in a pear tree.

The pattern 'ing' appears 7 times
```

COMPARING STRINGS

In Chapter 4 you learned that you can compare one string to another and branch to another place in the program based on the result of the comparison. The following IF statement compares the variable *reply$* to the literal string *Y* and prints a message based on the comparison:

```
reply$ = "Y"
IF (reply$ = "Y") THEN
    PRINT "The two string values are equal."
ELSE
    PRINT "The two string values are not equal."
END IF
```

If you execute the statements as they are above, you receive the following output:

```
The two string values are equal.
```

If you change the value of *reply$* to *y* and then execute the statement, you receive a different response:

```
The two string values are not equal.
```

Why is this? After all, a Y is a y.... Or is it? What criteria is QBasic using for its string comparisons?

The ASCII Character Set

Before QBasic can compare one character to another, it must convert each character into a number by using a translation table called the *ASCII character set.* QBasic then compares the numbers, called *ASCII codes*, and returns the logical value *true* if the ASCII codes are equal or *false* if the codes are not equal.

ASCII Is an Acronym

Like many other computer terms, ASCII is an acronym—it stands for American Standard Code for Information Interchange. The key word here is *code*: Like the Morse code used in radio and telegraphy, ASCII is an internationally accepted code for representing characters, but ASCII is used in computers and telecommunication. Appendix B lists the complete ASCII character set.

Each character in the ASCII character set is associated with a unique number; the set contains 128 characters (codes 0 through 127) in all:

- Control characters (codes 0 through 31), including characters that correspond to special keys on your keyboard such as Enter, Backspace, and Escape
- Punctuation symbols, numbers, and mathematical symbols (codes 32 through 64)
- Uppercase letters of the alphabet (codes 65 through 90)
- Lowercase letters of the alphabet (codes 97 through 122)
- Miscellaneous symbols (codes 91 through 96 and 123 through 127)

The ASCII code for the uppercase letter *A* is 65; the ASCII code for the lowercase letter *z* is 122. Following this logic, it's easy to see why QBasic considers the uppercase letter *Y* and the lowercase letter *y* to be different characters.

The IBM Extended Character Set

Appendix B contains an additional collection of characters (codes 128 through 255) known as the *IBM extended character set*. This set of symbols was developed by IBM for the IBM family of personal computers and has subsequently been adopted by most manufacturers of PC-compatible computers. The IBM extended character set (sometimes called the upper ASCII character set) contains foreign-language characters, line-drawing and box-drawing characters, and mathematical symbols. Although you can use these symbols in your programs, you can't type them by the usual means—they don't appear on your keyboard. Instead, you hold down the Alt key and type in the three-digit IBM extended character code. (Use the keys on the numeric keypad rather than the number keys near the top of your keyboard.) For example, to display the mathematical symbol π on your screen, hold down Alt and type *227* on the numeric keypad.

The following PRINT statement demonstrates a valid use of the IBM extended character set in a QBasic program. The ¿ symbol was entered by holding down Alt and typing *168*.

```
PRINT "¿Como se llama?"
```

NOTE: Some older monitors cannot display characters in the IBM extended character set, and you'll have trouble printing most of the IBM extended character set symbols unless your printer is specifically set up to handle them. Be ready for some strange (but nondamaging) results when you use the extended character set.

Converting ASCII Codes to Characters

If you know a character's ASCII code but you're not sure what the character it represents looks like, you can use the CHR$ function to return the symbol to your screen or to your program.

The syntax for the CHR$ function is as follows:

```
CHR$(code)
```

code is an integer value specifying an ASCII or IBM extended character code. The value returned by CHR$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

Practice: Using the CHR$ function

The ASCII.BAS program (Figure 9-18) shows how to use the CHR$ function to display the characters in the ASCII and IBM extended character sets. Note that the program skips the first 32 (control)

```
' ASCII.BAS
' This program displays the ASCII and IBM extended character sets.

CLS

FOR i% = 33 TO 255
    PRINT "Code"; i%; "= "; CHR$(i%)
    IF (i% MOD 23 = 0) THEN INPUT "Press Enter for more...", dummy$
NEXT i%
```

FIGURE 9-18.

ASCII.BAS: A program that uses the CHR$ function to display the ASCII and IBM extended character sets.

characters in the ASCII character set and pauses after each multiple of 23 lines (except the first and last sets) so that you can view the results at your own pace.

Type in the ASCII.BAS program and run it. (Because of the length of the program's output, we won't show it here.)

Converting Characters to ASCII Codes

As a complement to the CHR$ function, QBasic provides the ASC function, which converts a character to its code in the ASCII character set or the IBM extended character set.

The syntax of the ASC function is as follows:

```
ASC(stringexpression)
```

stringexpression is a one-character string. The integer value returned by ASC can be assigned to an integer variable or supplied as an argument to a statement or function that accepts integer values.

Practice: Using the ASC function

The COMPARE.BAS program (Figure 9-19) uses the ASC function to show how QBasic compares characters. COMPARE asks the user for two characters and then uses a SELECT CASE statement to display one of three messages based on the numeric values returned by ASC.

Type in the COMPARE.BAS program and run it.

```
' COMPARE.BAS
' This program compares any two characters.

CLS

INPUT "Enter any character:  ", firstChar$
INPUT "Enter another character:  ", secondChar$
PRINT
```

Figure 9-19. *(continued)*
COMPARE.BAS: A program that uses the ASC function to compare two characters.

FIGURE 9-19. *continued*

```
SELECT CASE ASC(firstChar$) - ASC(secondChar$)
    CASE IS < 0
        PRINT "'"; firstChar$; "' comes before '"; secondChar$; "'"
        PRINT "because"; ASC(firstChar$); "is less than"; ASC(secondChar$)
    CASE IS > 0
        PRINT "'"; firstChar$; "' comes after '"; secondChar$; "'"
        PRINT "because"; ASC(firstChar$); "is more than"; ASC(secondChar$)
    CASE ELSE
        PRINT "'"; firstChar$; "' is the same as '"; secondChar$; "'"
        PRINT "because"; ASC(firstChar$); "is equal to"; ASC(secondChar$)
END SELECT
```

You'll receive output similar to the following:

```
Enter any character:  d
Enter another character:  J

'd' comes after 'J'
because 100 is more than 74
```

Using Relational Operators with Strings

In addition to testing for equivalence of characters, QBasic supports string comparisons with the following relational operators:

Operator	*Meaning*	*Operator*	*Meaning*
<>	Not equal	>	Greater than
=	Equal	<=	Less than or equal to
<	Less than	>=	Greater than or equal to

A character is ''greater than'' another character if its ASCII code is higher. For example, the ASCII value of the letter *B* is greater than the ASCII value of the letter *A*, so the expression

`"A" < "B"`

is true, and the expression

`"A" > "B"`

is false.

When comparing two strings each of which contains more than one character, QBasic begins by comparing the first character in the first string to the first character in the other string and then proceeds through the strings character by character until it finds a difference. For example, the strings *Mike* and *Michael* are the same up to the third characters (*k* and *c*). Because the ASCII value of *k* is greater than that of *c*, the expression

```
"Mike" > "Michael"
```

is true.

If no differences are found, the strings are equal. If two strings are equal through several characters but one of the strings continues and the other one stops, the longer string is greater than the shorter string. For example, the expression

```
"AAAAA" > "AAA"
```

is true.

Sorting Strings: The STRSORT.BAS Program

The STRSORT.BAS program (Figure 9-20 on page 275) uses the <= relational operator to compare array elements and uses the SWAP statement to switch any elements that are out of order.

STRSORT declares an array of strings named *inputLines$* and calls the *GetText* subprogram. The *GetText* subprogram in STRSORT is identical to the *GetText* subprogram in the REPEAT.BAS program (Figure 9-17): It reads lines of text into an array and then returns the array (*inputLines$*) and the number of elements in the array (*numOfElements%*) to the main program.

STRSORT next calls the *ShellSort* subprogram. Note the following statements in the heart of *ShellSort* that compare array elements and then swap the elements if they are out of order:

```
IF strArray$(j%) <= strArray$(j% + span%) THEN EXIT FOR
' swap array elements that are out of order
SWAP strArray$(j%), strArray$(j% + span%)
```

The <= relational operator is used to compare the two array elements.

- If the value of the string expression *strArray$(j%)* is less than or equal to the value of the string expression *strArray$(j% + span%)*, the elements are already in order and the EXIT FOR statement terminates the FOR loop.
- If the value of the second string expression is greater than the value of the first, the elements are exchanged by means of the SWAP statement. SWAP, introduced here for the first time, exchanges the values of any two variables of the same type. In this case, the values of two string variables are exchanged. When the array is completely sorted, it is passed back to the main program and printed in its entirety by a FOR loop.

Practice: Using the <= relational operator and the SWAP statement

Type in the STRSORT.BAS program and run it. If you typed in the REPEAT.BAS program (Figure 9-17), you can use it as a foundation for STRSORT.BAS by deleting the *Repeat* function. (Press F2, highlight *Repeat*, and select Delete.) Next select the text in the main program and delete it. Then type in the *ShellSort* subprogram and the new main program. Save the new program as STRSORT.BAS.

The Shell Sort

The logic in the *ShellSort* subprogram is based on the popular Shell Sort algorithm for sorting an array of numbers published in 1959 by Donald Shell. The Shell Sort sorts a list of elements by continually dividing the main list into sublists that are smaller by half and comparing the elements at the tops and bottoms of the sublists. If the elements are out of order, they are exchanged. The end result is an array of items in descending order.

```
' STRSORT.BAS
' This program prompts the user for a list of names and then sorts the
'   names alphabetically.

' set maximum number of lines that can be entered and declare string
'   array to hold lines

CONST MAXLINES% = 15
DIM inputLines$(MAXLINES%)

' declare GetText and ShellSort subprograms

DECLARE SUB GetText (strArray$(), numOfElements%)
DECLARE SUB ShellSort (strArray$(), numOfElements%)

CLS

' call GetText subprogram to get input from user; at return the
'   numOfElements% variable will contain number of lines received

GetText inputLines$(), numOfElements%

' call ShellSort subprogram to put inputLines$ array in
'   alphabetical order

ShellSort inputLines$(), numOfElements%

PRINT
PRINT "Sorting results:"
PRINT

FOR i% = 1 TO numOfElements%    ' print contents of sorted array
   PRINT inputLines$(i%)
NEXT i%

END
```

FIGURE 9-20. *(continued)*

STRSORT.BAS: A program that sorts a list of strings entered by the user.

FIGURE 9-20. *continued*

```
SUB GetText (strArray$(), count%)

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard. The number of lines that can be
'   entered is determined by the global constant MAXLINES%.
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

PRINT "Enter up to"; MAXLINES%; "lines of text; to end, ";
PRINT "press Enter on a new line."
PRINT
count% = 0

DO
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN    ' if line is not blank, copy it
        count% = count% + 1    '     to the strArray$ array
        strArray$(count%) = inLine$
    END IF
' loop until count% = MAXLINES% or an empty line is received
LOOP WHILE (count% < MAXLINES%) AND (inLine$ <> "")

END SUB

SUB ShellSort (strArray$(), numOfElements%)

' The ShellSort subprogram sorts the elements of strArray$ and
'   returns strArray$ to the main program.  The numOfElements%
'   argument contains the number of elements in strArray$.
'   ShellSort sorts elements in descending order.

span% = numOfElements% \ 2
```

(continued)

FIGURE 9-20. *continued*

```
DO WHILE span% > 0
    FOR i% = span% TO numOfElements% - 1
        j% = i% - span% + 1
        FOR j% = (i% - span% + 1) TO 1 STEP -span%
            IF strArray$(j%) <= strArray$(j% + span%) THEN EXIT FOR
            ' swap array elements that are out of order
            SWAP strArray$(j%), strArray$(j% + span%)
        NEXT j%
    NEXT i%

    span% = span% \ 2
LOOP

END SUB
```

You'll receive output similar to the following:

```
Enter up to 15 lines of text; to end, press Enter on a new line.

-> Halvorson, Mike
-> Ullom, Kim
-> Halvorson, Ken
-> Zell, Linda
-> Halvorson, Victor
-> Zell, Ben
-> Berquist, Evelyn
-> Gullickson, Emma
->

Sorting results:

Berquist, Evelyn
Gullickson, Emma
Halvorson, Ken
Halvorson, Mike
Halvorson, Victor
Ullom, Kim
Zell, Ben
Zell, Linda
```

SUMMARY

You covered a lot of ground in this chapter and added many new functions and statements to your collection of programming tools. In all, you learned about 15 new statements and functions:

- UCASE$, LCASE$
- LEN
- RIGHT$, LEFT$, MID$
- RTRIM$, LTRIM$
- LINE INPUT
- SPACE$, STRING$
- INSTR
- CHR$, ASC
- SWAP

You also learned how to declare, combine, take apart, compare, and sort strings. QBasic provides such a variety of tools for working with strings because so much of what's done with personal computers revolves around working with large amounts of text data. The next chapter discusses efficient ways to manage all that data.

QUESTIONS AND EXERCISES

1. True or False: String constants are written in all-uppercase letters by convention.
2. True or False: The following declaration of a fixed-length string is a valid QBasic statement:

   ```
   DIM lastName$ AS STRING * 20
   ```

3. What output does the following QBasic statement produce?

   ```
   PRINT "ONE" + "TWO" + "THREE"
   ```

4. Which of the following values can be supplied as an argument to the LEN function?

   a. A literal string

   b. A string constant

   c. A string variable

   d. The result of a string function

5. What has happened if a call to the INSTR function returns the value 0?

6. What does the following QBasic statement display?

```
PRINT CHR$(ASC("h") - 32)
```

7. What is the ASCII code for the letter M?

8. Write a program that prompts the user for a first name and a last name, converts the names to uppercase, and then prints them in the format *Lastname, Firstname.*

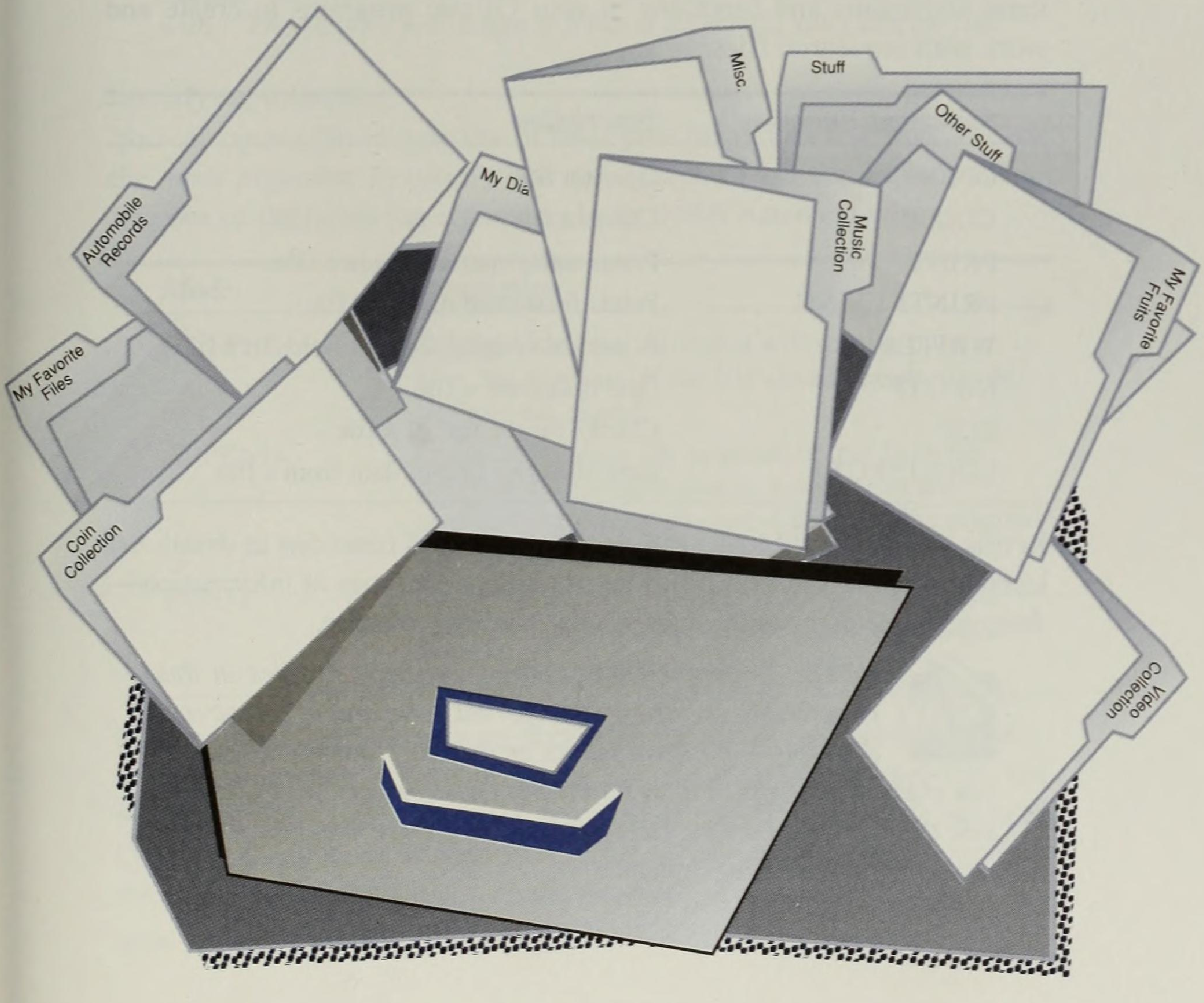
9. Write a program that gets a variable-length string from the user, reverses the order of the characters in the string, and displays the new string.

10. Write a program that gets a full, three-part name from the user with one variable-length string and copies each name in the string to a separate string variable. Your program should be able to process names of any length, provided that the user separates the three subnames with spaces. Output from the program should resemble the following:

```
Enter a string:  Queen Victoria Belfield

First name:  Queen
Middle name:  Victoria
Last name:  Belfield
```

CHAPTER 10

Working with Files


Automobile Records
My Dia
Misc.
Stuff
Other Stuff
Music Collection
My Favorite Fruits
My Favorite Files
Coin Collection
Video Collection

In Chapters 8 and 9 you learned how to manage large amounts of data in a program. In this chapter you'll learn how to save the data on disk and access it whenever you like. You'll also learn how to send program output to a printer.

CREATING AND USING SEQUENTIAL FILES

A *file* is simply a collection of data saved on disk. One type of file you can create from within a QBasic program is a *sequential file*. The contents of a sequential file must be used in order, from start to finish. (The contents of the other type of file you can create, a *random-access file,* can be used in any order, but we won't discuss random-access files in this book.) You use these statements and functions in your QBasic programs to create and work with sequential files:

Statement/Function	***Description***
OPEN	Opens a file
CLOSE	Closes a file
PRINT#	Prints unformatted data in a file
PRINT# USING	Prints formatted data in a file
WRITE#	Prints data organized into fields in a file
INPUT#	Gets data from a file
EOF	Checks for the end of a file
LINE INPUT#	Gets an entire line of data from a file

In this chapter you'll look at these statements and functions in detail and learn how to use sequential files to track different types of information—from a simple diary to the contents of an entire database.

NOTE: To simplify the learning process, the files in this chapter are created on the current drive and in the current directory—no drive letters or directory paths are used. If you know enough about drives and directories to work with files outside the current directory, feel free to do so. QBasic can take full advantage of the DOS file system.

Creating and Opening a File

The OPEN statement performs double duty: You use OPEN both to create new files and to open existing files. The syntax for the OPEN statement is as follows:

```
OPEN filename FOR mode AS #filenumber
```

filename is a valid DOS filename, *mode* is a word that indicates how the file is to be used, and *filenumber* is an integer from 1 through 255 to be associated with the file that is opened.

NOTE: Because you assign a number to your file, you can refer to it within your program simply by calling it by number. You should start numbering files within your programs with 1. The number you assign to a file is valid until you close the file.

Specifying a mode

You can open a file in only one of three possible modes at a time. You use the *mode* argument to specify how you intend to use the sequential file. Use one of the following arguments in an OPEN statement *mode*.

Mode	*Description*
OUTPUT	Creates and opens a file that will receive output from the program. If the file already exists, its old contents are erased.
APPEND	Opens an existing file to which output from the program is appended; that is, output from the program is placed at the *end* of the file. The original contents of the file are preserved.
INPUT	Opens a file that the program can only read. The program cannot change the file, but it can use the file's contents as input.

Some sample OPEN statements that show the use of each mode follow.

The following statement opens the NAMES.TXT sequential file, which will store output from the program. If NAMES.TXT does not exist, QBasic creates it. If it does exist, QBasic erases its existing contents. NAMES.TXT is associated with the number 1.

```
OPEN "NAMES.TXT" FOR OUTPUT AS #1
```

The following statement opens an existing file named NAMES.TXT and associates it with the number 1. The program stores its output at the end of the existing contents of NAMES.TXT.

```
OPEN "NAMES.TXT" FOR APPEND AS #1
```

The following statement opens an existing file named NAMES.TXT and associates it with the number 1. The program can use the contents of NAMES.TXT as input.

```
OPEN "NAMES.TXT" FOR INPUT AS #1
```

Using a string variable with OPEN

When using the OPEN statement, you can even use a string variable in place of *filename*. For example, you can prompt your user to supply a filename, and then you can assign the supplied name to a variable and use the variable within the OPEN statement, as shown in the following example:

```
INPUT "What file would you like to open?  ", filename$
OPEN filename$ FOR OUTPUT AS #1
```

You'll probably want to use some of the string-evaluation skills you learned in Chapter 9 to have the program verify that the user's response (*filename$*) is a valid DOS filename before it tries to open the file.

Closing a File

When you've finished working with a file, be sure to close it with the CLOSE statement. This ensures that all information you have written to the file is actually written to the disk. After a file is closed, the number associated with it is released and can be assigned to another file.

The syntax of the CLOSE statement is as follows:

```
CLOSE [#filenumber]
```

filenumber is the number associated with the file you want to close. If you omit the *#filenumber* argument, *all* open files are closed. Once you close a file, you must reopen it before you can use it again.

Let's look at some examples of the CLOSE statement. The following CLOSE statement closes file 1:

```
CLOSE #1
```

The following CLOSE statement closes all open files:

```
CLOSE
```

NOTE: Although QBasic automatically closes all open files when a program terminates, you should develop the habit of closing all files as soon as you are finished with them. Otherwise, an unforeseen event such as a power outage can cause your program to lose data.

Storing Data in a File

After a file is opened in OUTPUT or APPEND mode, it can receive data from your program. Three QBasic statements can send data to an open sequential file:

- The PRINT# statement sends *unformatted* data to a file.
- The PRINT# USING statement sends *formatted* data to a file.
- The WRITE# statement sends data organized into *fields* to a file.

The following sections describe each of these file-storage statements in detail.

The PRINT# statement

The PRINT# statement is functionally similar to the PRINT statement, but it sends data to a file rather than to the screen. The syntax for the PRINT# statement is as follows:

```
PRINT #filenumber,[ expressionlist][,¦;]
```

filenumber is the number of the open file, and *expressionlist* is the data to be sent to the file. If you omit *expressionlist*, a blank line is sent to the file.

If you put more than one item in *expressionlist*, you must separate the items with semicolons or commas. Semicolons and commas in the PRINT# statement work exactly as they do in the regular PRINT statement. If a semicolon is the last character in the PRINT# statement, the next item sent to the file appears at the end of the line you just sent.

Practice: Storing unformatted data in a file

The CARDATA.BAS program (Figure 10-1) demonstrates how to use the PRINT# statement to send three lines of unformatted data to a file named CARDATA.TXT. Values sent to a file can be literal values, simple variables, or the results of functions or expressions.

Type in the CARDATA.BAS program and run it.

```
' CARDATA.BAS
' This program uses the PRINT# statement to send three lines of
'   car information to a sequential file.

OPEN "CARDATA.TXT" FOR OUTPUT AS #1  ' open file in current drive/dir

CLS

' get some car information from the user and write it to the open file

INPUT "Enter the make of a car in your collection:  ", makeName$
INPUT "What is the model name? ", modelName$
INPUT "What year was the car made? ", year%

PRINT #1, makeName$, modelName$, year%

' add some literal values to the file

PRINT #1, "Mercedes-Benz", "190 SL", 1959

' add a new car to the file (RIGHT$ function returns current year
'   from DATE$ function)

PRINT #1, "Audi", "80 Quattro", " "; RIGHT$(DATE$, 4)

CLOSE #1                             ' close the file

PRINT
PRINT "Information has been successfully written to CARDATA.TXT"
```

FIGURE 10-1.
CARDATA.BAS: A program demonstrating the PRINT# statement.

You'll receive output similar to the following:

```
Enter the make of a car in your collection:  Ford
What is the model name? Thunderbird
What year was the car made? 1956

Information has been successfully written to CARDATA.TXT
```

A file named CARDATA.TXT is also created in the current directory on the disk in the current drive. To view the file, exit QBasic and type the following DOS command at the system prompt:

```
type cardata.txt
```

You'll receive output similar to the following:

```
Ford          Thunderbird    1956
Mercedes-Benz 190 SL         1959
Audi          80 Quattro     1991
```

The PRINT# USING statement

The PRINT# USING statement follows the same rules and uses the same special formatting characters as the PRINT USING statement, but it sends formatted data to a file rather than to the screen. PRINT# USING is helpful when you need to send large amounts of tabular data to a file. The syntax of the PRINT# USING statement is as follows:

```
PRINT #filenumber, USING template; [expressionlist][,|;]
```

filenumber is the number of the open file, *template* is a string used to format the values in *expressionlist*, and *expressionlist* is the data to be formatted and sent to the file. (You can use either comma or semicolon separators to separate the values in *expressionlist*.) The formatting characters in *template* must match up one-for-one with the characters of the values in *expressionlist*.

Practice: Storing formatted data in a file

The FRUITS.BAS program (Figure 10-2) demonstrates how to use the PRINT# USING statement to send three lines of formatted data to the FRUITS.TXT file. The formatting template in this program is the string variable *tmp$*, which uses several special formatting characters to align the data vertically before it is sent to the file.

Type in the FRUITS.BAS program and run it.

```
' FRUITS.BAS
' This program uses the PRINT# USING statement to send three lines
'   of formatted fruit information to a sequential file.

OPEN "FRUITS.TXT" FOR OUTPUT AS #1  ' open file in current drive/dir

CLS

' create a formatting template for the PRINT# USING statement

tmp$ = "Fruit:  \             \ Cases: ###    Price/pound:$$##.##"

' get some fruit information from the user and write it to the open file

INPUT "Enter your favorite summer fruit:  ", fruit$
INPUT "How many cases would you like to purchase?  ", cases%
INPUT "How much does it cost per pound?  $", cost!

PRINT #1, USING tmp$; fruit$; cases%; cost!

' add some literal values to the file

PRINT #1, USING tmp$; "Strawberry"; 2; 1.29

PRINT #1, USING tmp$; "Cantaloupe"; 14; .69

CLOSE #1                           ' close the file

PRINT
PRINT "Information has been successfully written to FRUITS.TXT"
```

FIGURE 10-2.
FRUITS.BAS: A program demonstrating the PRINT# USING statement.

You'll receive output similar to the following:

```
Enter your favorite summer fruit:  Pear
How many cases would you like to purchase?  10
How much does it cost per pound?  $0.89

Information has been successfully written to FRUITS.TXT
```

A file named FRUITS.TXT is also created in the current directory on the disk in the current drive. To view the file, exit QBasic and enter the following DOS command at the system prompt:

```
type fruits.txt
```

You'll receive output similar to the following:

```
Fruit:  Pear           Cases:  10   Price/pound:  $0.89
Fruit:  Strawberry     Cases:   2   Price/pound:  $1.29
Fruit:  Cantaloupe     Cases:  14   Price/pound:  $0.69
```

The WRITE# statement

The WRITE# statement does not format its output as the PRINT# and PRINT# USING statements do—WRITE# creates a file that will be read by other programs. (We'll use this statement later in the chapter when we create and track a database file.) The information that the WRITE# statement sends to a sequential file is separated by commas, and it's organized into groups called *fields.*

The syntax for the WRITE# statement is as follows:

```
WRITE #filenumber[, expressionlist]
```

filenumber is the number of the open file, and *expressionlist* is the data to be sent to the file. Elements in *expressionlist* are separated by commas. If you omit *expressionlist*, a blank line is written to the file.

Figure 10-3 shows a sample WRITE# statement that sends five types of values to a file. Note how the commas between values separate the output into fields. Also note the string value surrounded by quotation marks: If the string contained spaces or commas, a non-QBasic program would recognize the spaces or commas as part of the string. If the file created with the WRITE# statement contains multiple lines, each line is called a *record.*

Sample program statements:

```
OPEN "TEST.TXT" FOR OUTPUT AS #1—— OPEN statement

a$ = "Kimberly"
b% = 25
c& = 100000         ——————————————— Sample variables
d! = 115.5
e# = .0123456789#

WRITE #1, a$, b%, c&, d!, e#———— WRITE# statement
CLOSE #1———————————————————————— CLOSE statement
```

Output sent to file:

```
"Kimberly", 25, 100000, 115.5, .0123456789
    |        |    |      |         |
 Field 1  Field 2 Field 3 Field 4   Field 5
```

FIGURE 10-3.
The WRITE# statement stores data in a file in fields and records.

Practice: Storing data in a file with fields

The COINS.BAS program (Figure 10-4) demonstrates how to use the WRITE# statement to store coin-collection information in a file named COINS.TXT. COINS.BAS is designed to let the user enter information about as many coins as desired. To stop entering collection information, the user simply types *END* when prompted for a country.

Type in the COINS.BAS program and run it.

```
' COINS.BAS
' This program uses the WRITE# statement to send coin-collection
'   information to a sequential file in fields.

OPEN "COINS.TXT" FOR OUTPUT AS #1  ' open file in current drive/dir

CLS
```

FIGURE 10-4. *(continued)*
COINS.BAS: A program demonstrating the WRITE# statement.

FIGURE 10-4. *continued*

```
PRINT "This program stores coin-collection information on disk in a"
PRINT "file named COINS.TXT.  Enter coin data and type END to quit."
PRINT
DO WHILE (country$ <> "END")       ' until the user types END...

' get coin-collection info from user and write it to the open file

    INPUT "What country is the coin from?  ", country$
    IF (country$ <> "END") THEN    ' if country$ is END don't write
        INPUT "What is the value of the coin?  ", value$
        INPUT "What is the name of the coin?   ", coinName$
        INPUT "What year was the coin minted?  ", year%

        WRITE #1, country$, value$, coinName$, year%  ' send fields
    END IF
    PRINT                          ' print blank lines between coins

LOOP

CLOSE #1                           ' close the file

PRINT "Information has been successfully written to COINS.TXT"
```

You'll receive output similar to the following:

```
This program stores coin-collection information on disk in a
file named COINS.TXT.  Enter coin data and type END to quit.

What country is the coin from?  United States
What is the value of the coin?  10 cents
What is the name of the coin?   Dime
What year was the coin minted?  1980

What country is the coin from?  Canada
What is the value of the coin?  25 cents
What is the name of the coin?   Quarter
What year was the coin minted?  1960
```

(continued)

continued

```
What country is the coin from?  Hungary
What is the value of the coin?  2 forints
What is the name of the coin?
What year was the coin minted?  1985

What country is the coin from?  Great Britain
What is the value of the coin?  1 pound
What is the name of the coin?   Pound
What year was the coin minted?  1981

What country is the coin from?  END

Information has been successfully written to COINS.TXT
```

A file named COINS.TXT is also created in the current directory on the disk in the current drive. To view the file, exit QBasic and type the following DOS command at the system prompt:

```
type coins.txt
```

You'll receive output similar to the following:

```
"United States","10 cents","Dime",1980
"Canada","25 cents","Quarter",1960
"Hungary","2 forints","",1985
"Great Britain","1 pound","Pound",1981
```

The COINS.TXT file contains four records of four fields each, as shown in Figure 10-5:

```
Record 1 ——"United States","10 cents","Dime",1980
Record 2 ——"Canada","25 cents","Quarter",1960
Record 3 ——"Hungary","2 forints","",1985
Record 4 ——"Great Britain","1 pound","Pound",1981
```

FIGURE 10-5.
Each line in the COINS.TXT file is a record containing four fields.

Getting Data from a File

Now that you've created three files and used the DOS command TYPE to examine them, you'll learn how to examine and use data files from within a QBasic program. You'll examine two statements and one function that help you work with files that have been opened for input:

- The INPUT# statement gets *one or more fields* of data from a file.
- The EOF function determines whether the end of the file has been reached.
- The LINE INPUT# statement gets an *entire line* of data from a file.

The INPUT# statement

The INPUT# statement is the primary tool you use to get data from a QBasic sequential file. The INPUT# statement gets input from a sequential file in much the same way that the INPUT statement gets input from the keyboard: Both statements assign one or more data items to variables of matching types. The syntax of the INPUT# statement is as follows:

```
INPUT #filenumber, variablelist
```

filenumber is the number of the open file, and *variablelist* is one or more variables to be assigned data from the file. Items in *variablelist* must be of the same type as items in the sequential file. Items in the sequential file can be fields of data created with the WRITE# statement or output from the PRINT# statement, from the PRINT# USING statement, or from any program that can create data files.

Practice: Getting data items from a sequential file

The FRIENDS.BAS program (Figure 10-6) demonstrates how to use the INPUT# statement in your QBasic program to get data from a sequential file. The program first opens a sequential file in the current directory for output and sends four strings to it with the WRITE# statement. The program then closes the file, opens it again for input, and reads the four strings back into the program.

Type in the FRIENDS.BAS program and run it.

```
' FRIENDS.BAS
' This program demonstrates the INPUT# statement.

CLS

OPEN "FRIENDS.TXT" FOR OUTPUT AS #1   ' open file for output

PRINT "Enter the names of four of your friends"
PRINT

FOR i% = 1 TO 4                       ' loop 4 times
    INPUT "Friendly name:  ", pal$    ' each time get name from user
    WRITE #1, pal$                    '   and write it to disk
NEXT i%

CLOSE #1                              ' close the file

OPEN "FRIENDS.TXT" FOR INPUT AS #1    ' reopen the file for input

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO 4                       ' loop 4 times
    INPUT #1, pal$                    ' each time get name from file
    PRINT pal$                        '   and display it on screen
NEXT i%

CLOSE #1                              ' close the file
```

FIGURE 10-6.
FRIENDS.BAS: A program demonstrating the INPUT# statement.

You'll receive output similar to the following:

```
Enter the names of four of your friends

Friendly name:  Larry
Friendly name:  Whitey
Friendly name:  Gus
Friendly name:  Gilbert
```

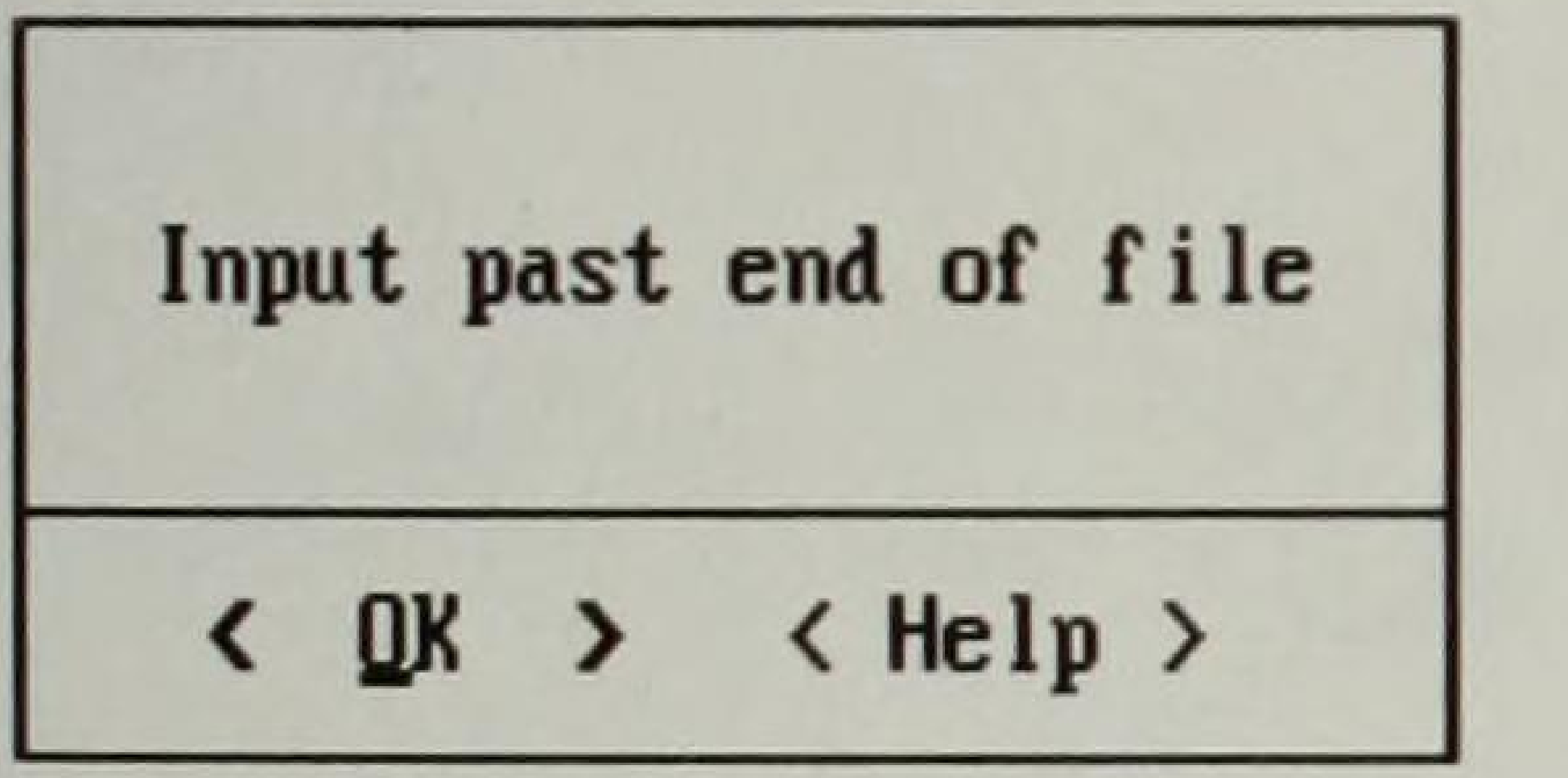
```
You entered the following names:

Larry
Whitey
Gus
Gilbert
```

The file FRIENDS.TXT is also created on disk and contains the following data records:

```
"Larry"
"Whitey"
"Gus"
"Gilbert"
```

The EOF function

If your QBasic program can't tell where your sequential file ends, you might receive an end-of-file error message. This error occurs when QBasic tries to read beyond the end of the file.



Input past end of file

To prevent this error, you must explicitly tell QBasic to quit reading values after it encounters the end of the file. To do this, use the EOF function. EOF has the following syntax:

```
EOF(filenumber)
```

filenumber is the number of the open file you want to check.

EOF returns the logical value *true* if the next character to be read is past the end of the file, and *false* otherwise.

You can use the EOF function anywhere in your program to check the status of an open file, but it is most effective in a DO loop, as shown in the following practice session.

Practice: Detecting the end of a file

The COINS2.BAS program (Figure 10-7) is an enhancement of the COINS.BAS program we worked with earlier in this chapter: It uses the INPUT# statement to display file data and the EOF function to check for the end of the file. This new version opens the coin-collection file in APPEND mode so that coin-collection information gathered in previous runs of the program isn't overwritten by the new information; instead, all new information is placed at the end of the file.

Type in the COINS2.BAS program and run it.

```
' COINS2.BAS
' This program uses the WRITE# statement to send coin-collection
'   information to a sequential file and INPUT# to display it.

' open file in APPEND mode so that previous contents won't be overwritten

OPEN "COINS.TXT" FOR APPEND AS #1  ' open file in current drive/dir

CLS

PRINT "This program stores coin-collection information on disk in a"
PRINT "file named COINS.TXT.  Enter coin data and type END to quit."
PRINT

DO WHILE (country$ <> "END")        ' until the user types END...

' get coin-collection info from user and write it to the open file

    INPUT "What country is the coin from? ", country$
    IF (country$ <> "END") THEN    ' if country$ is END don't write
        INPUT "What is the value of the coin? ", value$
        INPUT "What is the name of the coin?  ", coinName$
        INPUT "What year was the coin minted? ", year%

        WRITE #1, country$, value$, coinName$, year%  ' send fields
    END IF
```

FIGURE 10-7. *(continued)*
COINS2.BAS: A program demonstrating the EOF function and the INPUT# statement.

FIGURE 10-7. *continued*

```
    PRINT                          ' print blank lines between coins

LOOP

CLOSE #1                           ' close the file

' wait for the user to press Enter to continue

INPUT "Press Enter to see the contents of your coin collection", dummy$
CLS                                ' start with a fresh screen

' open file in INPUT mode so that contents can be read by program

OPEN "COINS.TXT" FOR INPUT AS #1   ' file is in current drive/dir

' display header for tabular collection information

PRINT "Coin origin      Coin value    Coin name     Year minted"
PRINT "--------------------------------------------------------"
PRINT

' initialize formatting template for use with PRINT USING

tmp$ = "\             \  \          \  \          \  ####"

' while the end of the file has not been reached, assign file
'   items to variables and print them out

DO WHILE (NOT EOF(1))
    INPUT #1, country$, value$, coinName$, year%
    PRINT USING tmp$; country$; value$; coinName$; year%
LOOP

CLOSE #1                           ' close the file
```

You'll be prompted for input in the following manner:

```
This program stores coin-collection information on disk in a
file named COINS.TXT.  Enter coin data and type END to quit.

What country is the coin from?  Netherlands
What is the value of the coin?  2.5 guilders
What is the name of the coin?   Rijksdaalder
What year was the coin minted?  1982

What country is the coin from?  END
```

After you've entered the new information, it is displayed along with the information already in the file:

```
Coin origin      Coin value    Coin name     Year minted
--------------------------------------------------------


United States    10 cents      Dime          1980
Canada           25 cents      Quarter       1960
Hungary          2 forints                   1985
Great Britain    1 pound       Pound         1981
Netherlands      2.5 guilders  Rijksdaalder  1982
```

The LINE INPUT# statement

The INPUT# statement is an effective way to obtain individual items from files, but what do you do if you need to obtain long strings of textual information? QBasic provides the LINE INPUT# statement for this purpose. LINE INPUT# is similar to LINE INPUT; it simply obtains input from a file instead of from a keyboard. LINE INPUT# has the following syntax:

```
LINE INPUT #filenumber, variablename$
```

filenumber is the number of the open file to be read from, and *variablename$* is the string variable that is to receive the line of input.

The LPRINT Statement

When you want to send information to the printer attached to your computer, use the LPRINT statement. LPRINT has a syntax similar to that of the PRINT statement:

```
LPRINT [expressionlist][,¦;]
```

expressionlist is a list of numeric or string expressions separated by commas or semicolons. For example, to send the string *Think Snow!* to the printer, use the following LPRINT statement:

```
LPRINT "Think Snow!"
```

To send a formfeed (page eject) command to your printer when you've finished printing, use the following statement:

```
LPRINT CHR$(12)
```

This sends ASCII character 12 to the printer to advance the paper.

Practice: Reading a diary with LINE INPUT#

The DIARY.BAS program (Figure 10-8) demonstrates how to use the LINE INPUT and LINE INPUT# statements together to track lines of textual material in a sequential file. DIARY.BAS is first opened in APPEND mode and is then marked with the current time and date from the system clock. The user can then add one or more lines to the diary, which is stored in the DIARY.TXT file in the current directory. Because the diary file is opened in APPEND mode, each entry is added to the end of the file, preserving the earlier entries. If the user requests a printout, all entries (including their time and date stamps) are sent to the printer by means of LPRINT statements.

Type in the DIARY.BAS program and run it.

```
' DIARY.BAS
' This program maintains a computer diary in a sequential file
'   named DIARY.TXT.

CLS

OPEN "DIARY.TXT" FOR APPEND AS #1   ' open file in append mode

PRINT "Enter your secret thoughts for today; type END to quit."
PRINT

PRINT #1, TIME$; "  "; DATE$        ' write time and date to file
PRINT #1,                           ' write blank line to file

DO WHILE (UCASE$(line$) <> "END")   ' until user types "END",
    LINE INPUT line$                '   get lines of text
    IF (line$ <> "END") THEN PRINT #1, line$
LOOP                                '   and write them to the file

PRINT #1,                           ' write blank line to file
CLOSE #1                            ' close file

PRINT                               ' find out if user wants a printout
INPUT "Would you like to print out your entire diary (Y,N)?  ", reply$

IF (UCASE$(reply$) = "Y") THEN        ' if yes,
    OPEN "DIARY.TXT" FOR INPUT AS #1  '   open file for input

    LPRINT STRING$(33, "-");        ' print a header at top of page
    LPRINT " My Diary ";
    LPRINT STRING$(33, "-")
    LPRINT                          '   and a blank line

    DO WHILE (NOT EOF(1))           ' until end of file is reached,
       LINE INPUT #1, line$         '   read lines from file
       LPRINT line$                 '   and send them to printer
    LOOP
```

FIGURE 10-8. *(continued)*
DIARY.BAS: A simple diary program that demonstrates the LINE INPUT# statement.

FIGURE 10-8. *continued*

```
    CLOSE #1                        ' close file

    LPRINT CHR$(12)                 ' send formfeed character

    PRINT                           ' display message indicating diary
    PRINT "Diary sent to printer"   '   contents have been sent to
END IF                              '   printer
```

When you run the program, you are prompted for a diary entry in the following manner:

```
Enter your secret thoughts for today; type END to quit.
```

After you type in your entry and the word *END*, you'll receive the following prompt:

```
Would you like to print out your entire diary (Y,N)?
```

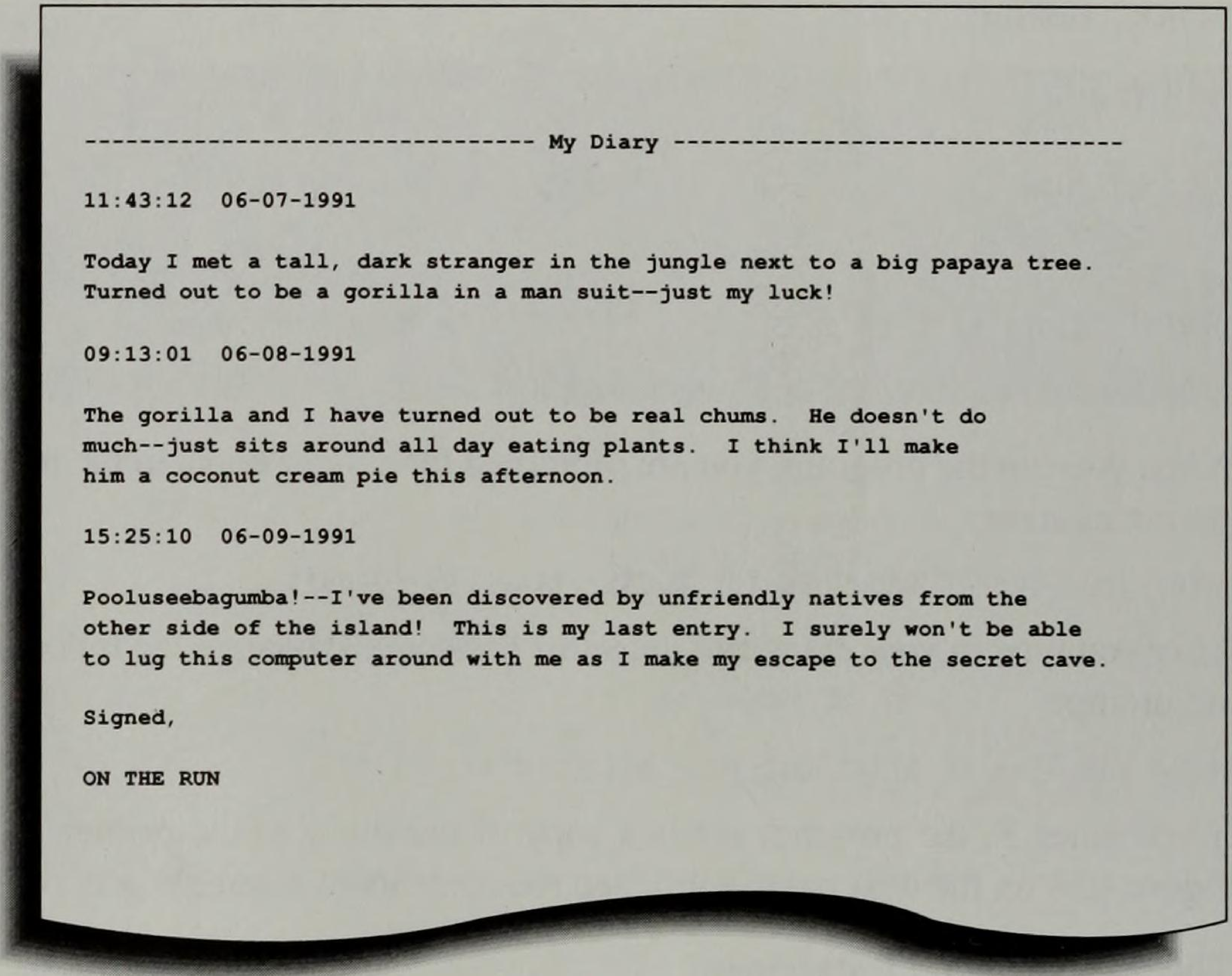
If you enter *Y*, the program sends a copy of the diary to the printer. In Figure 10-9 on the next page, you'll see the contents of a sample printout.

The VIEW PRINT statement

The last statement we'll discuss in this section is VIEW PRINT. VIEW PRINT isn't used directly with files but is sometimes used in combination with the INPUT#, LOCATE, and PRINT statements to display the contents of a file in different places on the screen. So far we've used the entire screen for the output of statements such as PRINT. When the 25 lines of the screen have filled up, additional PRINT statements cause the screen to scroll. With the VIEW PRINT statement, however, you can restrict printed output to only part of the screen—called a *text viewport*—while the rest remains intact. The syntax for the VIEW PRINT statement is as follows:

```
VIEW PRINT [topline TO bottomline]
```

topline is the top line of the text viewport, and *bottomline* is the bottom line of the text viewport. If *topline* and *bottomline* are not included, the entire display becomes the text viewport (the default).



```
-------------------------------- My Diary --------------------------------

11:43:12  06-07-1991

Today I met a tall, dark stranger in the jungle next to a big papaya tree.
Turned out to be a gorilla in a man suit--just my luck!

09:13:01  06-08-1991

The gorilla and I have turned out to be real chums.  He doesn't do
much--just sits around all day eating plants.  I think I'll make
him a coconut cream pie this afternoon.

15:25:10  06-09-1991

Pooluseebagumba!--I've been discovered by unfriendly natives from the
other side of the island!  This is my last entry.  I surely won't be able
to lug this computer around with me as I make my escape to the secret cave.

Signed,

ON THE RUN
```

FIGURE 10-9.
A printout produced by the DIARY.BAS program.

Once a viewport has been established, scrolling takes place only between the top and bottom lines of the viewport. This is very convenient when you want to display a list of instructions or table headings and you don't want them to scroll off the screen. Note that when a viewport is in effect you can use the statement

```
CLS 2
```

to clear only the viewport portion of the screen and leave the rest intact.

Practice: Creating a text viewport

The VIEW.BAS program (Figure 10-10) demonstrates the operation of the VIEW PRINT, LOCATE, PRINT, and CLS statements. We'll see more of these statements in the database program near the end of the chapter.

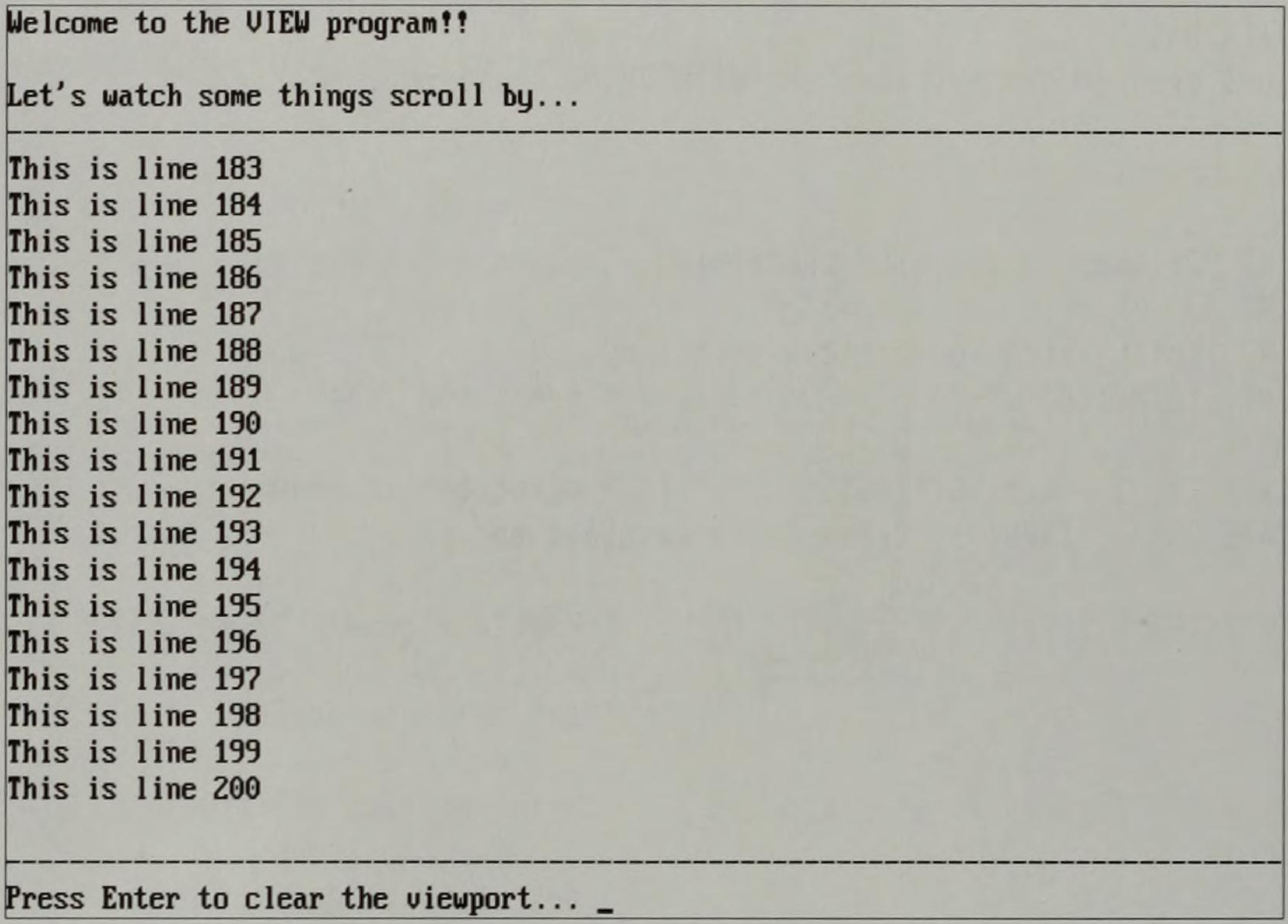
```
' VIEW.BAS
' This program demonstrates the VIEW PRINT statement.

CLS

PRINT "Welcome to the VIEW program!!"  ' print intro message
PRINT
PRINT "Let's watch some things scroll by..."
PRINT STRING$(80, "-");                ' print top line

LOCATE 24, 1: PRINT STRING$(80, "-");  ' print bottom line
LOCATE 25, 1: INPUT ; "Press Enter to start the program... ", dummy$

VIEW PRINT 5 TO 23                     ' enable viewport (lines 5-23)

COLOR 2                                ' set color to green
FOR i% = 1 TO 200
    PRINT "This is line"; i%           ' print message 200 times
NEXT i%
COLOR 7                                ' set color to default white

VIEW PRINT                             ' disable viewport
LOCATE 25, 1: INPUT ; "Press Enter to clear the viewport... ", dummy$
VIEW PRINT 5 TO 23                     ' enable viewport
CLS 2                                  ' clear viewport only
```

FIGURE 10-10.
VIEW.BAS: A program demonstrating the VIEW PRINT statement.

Type in the VIEW.BAS program and run it. You'll see a screen with a divided viewport, as shown at the top of the next page.

QBasic Equivalents of DOS Commands

Many QBasic statements have counterparts in the DOS world. If you're familiar with file-related DOS commands, you might find Table 10-1 useful. It lists QBasic statements along with their DOS-command counterparts and descriptions of their purposes.



```
Welcome to the VIEW program!!

Let's watch some things scroll by...
--------------------------------------------------------------------------------
This is line 183
This is line 184
This is line 185
This is line 186
This is line 187
This is line 188
This is line 189
This is line 190
This is line 191
This is line 192
This is line 193
This is line 194
This is line 195
This is line 196
This is line 197
This is line 198
This is line 199
This is line 200

--------------------------------------------------------------------------------
Press Enter to clear the viewport... _
```

QBasic statement	*DOS equivalent*	*Description*
FILES	DIR/W	Displays a directory listing
SHELL	COMMAND	Runs a DOS command
NAME	RENAME	Changes the name of a file
KILL	DEL *or* ERASE	Deletes a file
MKDIR	MKDIR	Creates a directory
CHDIR	CHDIR	Changes the current directory
RMDIR	RMDIR	Removes an empty directory

TABLE 10-1.
QBasic equivalents of DOS commands.

The FILES statement

If you want your program to display a list of files from which a user can choose, use the FILES statement. FILES has the following syntax:

```
FILES [filename]
```

filename is the optional filename of the file you want to see a directory listing for. If you omit *filename*, QBasic displays a listing of all files in the current directory. You can use the * and *?* wildcard characters as part of *filename* in order to display a wide range of files. To display a listing for a directory or drive other than the current directory, include a drive letter or a pathname or both as part of *filename*.

Practice: Viewing the current directory

The SHOW.BAS program (Figure 10-11) demonstrates how to use the FILES command to display all files in the current directory whose names end with *.TXT*. After a directory listing is displayed, SHOW.BAS prompts the user for a filename and prints out the specified text file.

Type in the SHOW.BAS program and run it.

```
' SHOW.BAS
' This program lets you view a data file in the current directory.

CLS

PRINT "The current directory contains the following files with ";
PRINT "a .TXT extension:"
PRINT

FILES "*.TXT"

PRINT
INPUT "Which file would you like to display?  ", filename$

OPEN filename$ FOR INPUT AS #1

PRINT
PRINT "---------- "; UCASE$(filename$); " ----------"
PRINT
```

FIGURE 10-11. *(continued)*
SHOW.BAS: A program demonstrating the FILES statement.

FIGURE 10-11. *continued*

```
DO WHILE (NOT EOF(1))
    LINE INPUT #1, line$
    PRINT line$
LOOP
```

When you run the program you get output similar to the following:

```
The current directory contains the following files with a .TXT extension:

C:\DOS
CARDATA .TXT       FRUITS  .TXT       COINS   .TXT       FRIENDS  .TXT
DIARY   .TXT
 2080768 Bytes free

Which file would you like to display?  fruits.txt

---------- FRUITS.TXT ----------

Fruit:  Pear            Cases:  10    Price/pound:  $0.89
Fruit:  Strawberry      Cases:   2    Price/pound:  $1.29
Fruit:  Cantaloupe      Cases:  14    Price/pound:  $0.69
```

The SHELL statement

The SHELL statement lets you exit QBasic temporarily so that you can run DOS commands. The SHELL statement is useful when you want to execute a command not available in QBasic or when you want to give users access to DOS while a program is running. (This feature is provided by many programs on the market today. In Microsoft Windows, for example, you can exit to DOS temporarily by selecting the MS-DOS Prompt icon.)

The syntax for the SHELL statement is as follows:

```
SHELL [commandString]
```

commandString is an optional string value or string variable containing a valid DOS command. If you include *commandString*, DOS executes the command and returns to the QBasic program. If you omit *commandString*, DOS runs a new temporary copy of itself. In nearly all respects this temporary copy looks and operates just as the primary copy of DOS does—you can enter any number of DOS commands. This copy of DOS is only temporary, however: When you type *exit* and press Enter, the QBasic program continues executing.

Practice: Running a DOS command

The RUNDOS.BAS program (Figure 10-12) demonstrates how to use the SHELL statement to run DOS commands from within a QBasic program. To help you track what's displayed by the program and what's displayed by DOS, the program begins by setting the foreground color to cyan. This ensures that the results of QBasic statements will appear in cyan and that the results of DOS commands will appear in white, the system default.

1. Type in the RUNDOS.BAS program and run it.

```
' RUNDOS.BAS
' This program executes a DOS command.

CLS

COLOR 3                     ' set color to cyan
                            ' get DOS command line from user
INPUT "Enter a DOS command line [press Enter to run DOS]:  ", dosCom$
PRINT

SHELL dosCom$               ' exit to DOS and execute the command

PRINT
PRINT "DOS command complete"

COLOR 7                     ' set color to default white
```

FIGURE 10-12.
RUNDOS.BAS: A program demonstrating the SHELL statement.

QBasic displays the following prompt:

```
Enter a DOS command line [press Enter to run DOS]:
```

2. To test how the SHELL statement processes one DOS command, type *chkdsk* and press Enter. After a few moments DOS displays information similar to the following:

```
Volume MIKE        created 02-19-1990 4:36p
Volume Serial Number is 1810-1496

  21690368 bytes total disk space
     77824 bytes in 6 hidden files
    120832 bytes in 54 directories
  19410944 bytes in 1003 user files
   2080768 bytes available on disk

      2048 bytes in each allocation unit
     10591 total allocation units on disk
      1016 available allocation units on disk

    655360 total bytes memory
    361888 bytes free

DOS command complete
```

   The SHELL statement processed the DOS CHKDSK command and then returned to the program.

3. Now run the RUNDOS program again, but this time simply press Enter to run a temporary version of DOS. You'll see something similar to the following output. Note the way we used our time in DOS to copy a file.

```
Enter a DOS command line [press Enter to run DOS]:


Microsoft(R) MS-DOS(R) Version 5.00
             (C)Copyright Microsoft Corp 1981-1990.

C:\DOS>dir *.txt

 Volume in drive C is MIKE
 Volume Serial Number is 1810-1496
 Directory of C:\DOS
```

```
CARDATA  TXT       107 06-06-91  12:40p
FRUITS   TXT       177 06-06-91   3:56p
COINS    TXT       197 06-06-91   5:34p
FRIENDS  TXT        37 06-07-91  10:19a
DIARY    TXT       713 06-07-91   3:12p
        5 file(s)     1231 bytes
                   2080768 bytes free
C:\DOS>copy diary.txt a:
        1 File(s) copied

C:\DOS>exit

DOS command complete
```

The NAME statement

The NAME statement is useful when you want to rename a file or move a file from one directory to another. The syntax of the NAME statement is as follows:

```
NAME oldFileName AS newFileName
```

oldFileName is the current filename, and *newFileName* is the new name you'd like the file to have. You can use directory pathnames in both *oldFileName* and *newFileName*. If the two pathnames are different, QBasic moves the file to the directory specified by *newFileName*. You can't move files between different drives, however. If the NAME statement cannot rename the files, the program generates an error message.

Practice: Renaming a file

The NEWNAME.BAS program (Figure 10-13) demonstrates how to use the NAME statement to change the name of a file in the current drive. Note the use of the FILES statement, which displays the files in the current directory.

Type in the NEWNAME.BAS program and run it.

```
' NEWNAME.BAS
' This program lets you rename a data file in the current directory.

CLS

PRINT "The current directory contains the following files:"
PRINT

FILES "*.*"                ' display files in current directory

PRINT                      ' get old and new filenames
INPUT "Which file would you like to rename?  ", oldName$
INPUT "What would you like the new name to be?  ", newName$

NAME oldName$ AS newName$  ' try to rename file

' if oldName$ does not exist or newName$ is an invalid name, the
'   NAME statement will generate a run-time error; otherwise, the
'   following lines will be executed:

PRINT                      ' print success message
PRINT UCASE$(oldName$); " renamed successfully"
```

FIGURE 10-13.
NEWNAME.BAS: A program demonstrating the NAME statement.

When you run the program, you receive output similar to the following:

```
The current directory contains the following files:

C:\DOS
        .   <DIR>         ..  <DIR> FRUITS  .BAS          CARDATA .BAS
CARDATA .TXT      FRUITS  .TXT      COINS   .BAS          COINS   .TXT
COINS2  .BAS      FRIENDS .BAS      FRIENDS .TXT          DIARY   .BAS
SHOW    .BAS      NEWNAME .BAS      DIARY   .TXT          RUNDOS  .BAS

 2080013 Bytes free

Which file would you like to rename?  diary.txt
What would you like the new name to be?  olddiary.txt

DIARY.TXT renamed successfully
```

The KILL statement

The KILL statement is both straightforward and permanent: It deletes a file from disk. Once a file is deleted by means of the KILL statement, it cannot be recovered. Programmers commonly use KILL to delete temporary files created during program execution, but you can also use KILL for general disk cleanup. The syntax of the KILL statement is as follows:

```
KILL filename
```

filename is the name of the file to be deleted. You can also use drive letters, pathnames, and the * and ? wildcard characters within a KILL statement.

NOTE: Use extreme caution with the KILL statement—especially when using wildcard characters. You might accidentally delete more files than you intend to. The following practice session shows some of the safety features you should build into a program that uses the KILL statement.

Practice: Deleting a file

The KILL.BAS program (Figure 10-14) demonstrates how to use the KILL statement to delete an unwanted file. If the user includes wildcard characters in the filename entered for deletion, the program detects them with INSTR functions and displays a warning message in red, asking the user to verify the multiple deletion. If the file specified for deletion does not exist, the KILL statement generates a run-time error message.

Type in the KILL.BAS program and run it.

```
' KILL.BAS
' This program lets you delete a file in the current directory.

CLS

PRINT "The current directory contains the following files:"
PRINT

FILES "*.*"                 ' display files in current directory
```

FIGURE 10-14. *(continued)*
KILL.BAS: A program demonstrating the KILL statement.

FIGURE 10-14. *continued*

```
PRINT                      ' get file to delete
INPUT "Which file(s) would you like to delete?  ", filename$

' check for wildcard characters (? and *) in filename$
'   if they exist, print a warning before deleting multiple files
'   if they do not exist, proceed with deletion
'   if the file does not exist, KILL will display a run-time error

IF (INSTR(filename$, "?")) OR (INSTR(filename$, "*")) THEN
    PRINT
    COLOR 4                ' set color to red for effect
    PRINT "Danger:  Wildcards detected that can delete multiple files!"
    COLOR 7                ' set color to default white
    PRINT                  ' prompt for deletion confirmation
    INPUT "Do you want to proceed (Y,N)?  ", reply$
                           ' if reply$ is yes, then delete file
    IF (UCASE$(reply$) = "Y") THEN
        KILL filename$
        PRINT
        PRINT UCASE$(filename$); " deleted from system"
    END IF                 ' ...if not, then exit program
ELSE
    KILL filename$         ' if no wildcards exist, simply delete file
    PRINT
    PRINT UCASE$(filename$); " deleted from system"
END IF
```

You'll receive output similar to the following:

```
The current directory contains the following files:

C:\DOS
        .   <DIR>           ..  <DIR> FRUITS  .BAS      CARDATA .BAS
CARDATA .TXT      FRUITS  .TXT      COINS   .BAS      COINS   .TXT
COINS2  .BAS      FRIENDS .BAS      FRIENDS .TXT      DIARY   .BAS
SHOW    .BAS      NEWNAME .BAS      OLDDIARY.TXT      KILL    .BAS
RUNDOS  .BAS
 2078419 Bytes free
```

```
Which file(s) would you like to delete? olddiary.txt

OLDDIARY.TXT deleted from system
```

TRACKING A DATABASE WITH SEQUENTIAL FILES

We'll conclude this chapter with a look at a simple database program, MUSICDB.BAS, that uses many of the statements and functions we've discussed to work with data stored in a sequential file. MUSICDB.BAS tracks the following information for each record, cassette, and compact disc in your collection:

- Title
- Recording artist
- Year released
- Style of music (rock, blues, jazz, classical, etc.)
- Medium (record, cassette, compact disc, etc.)

MUSICDB.BAS also provides a number of ''database style'' features to help you monitor your collection. MUSICDB.BAS lets you

- Store music-collection information permanently within a sequential file.
- Examine the music-collection database records one at a time on screen.
- Print the entire music-collection database.
- Search for database records by title or recording artist.
- Load and examine other music-collection databases.
- Exit temporarily to DOS to run DOS commands.

A sizable amount of MUSICDB.BAS code is devoted to the menu-driven user interface of the program, which should give you a feel for how screens typical of general-purpose application programs are created. MUSICDB.BAS is general enough that you can easily modify it to track other types of database information. We'll discuss how you can do this after we look at how MUSICDB.BAS operates.

Tracking Information with a Database

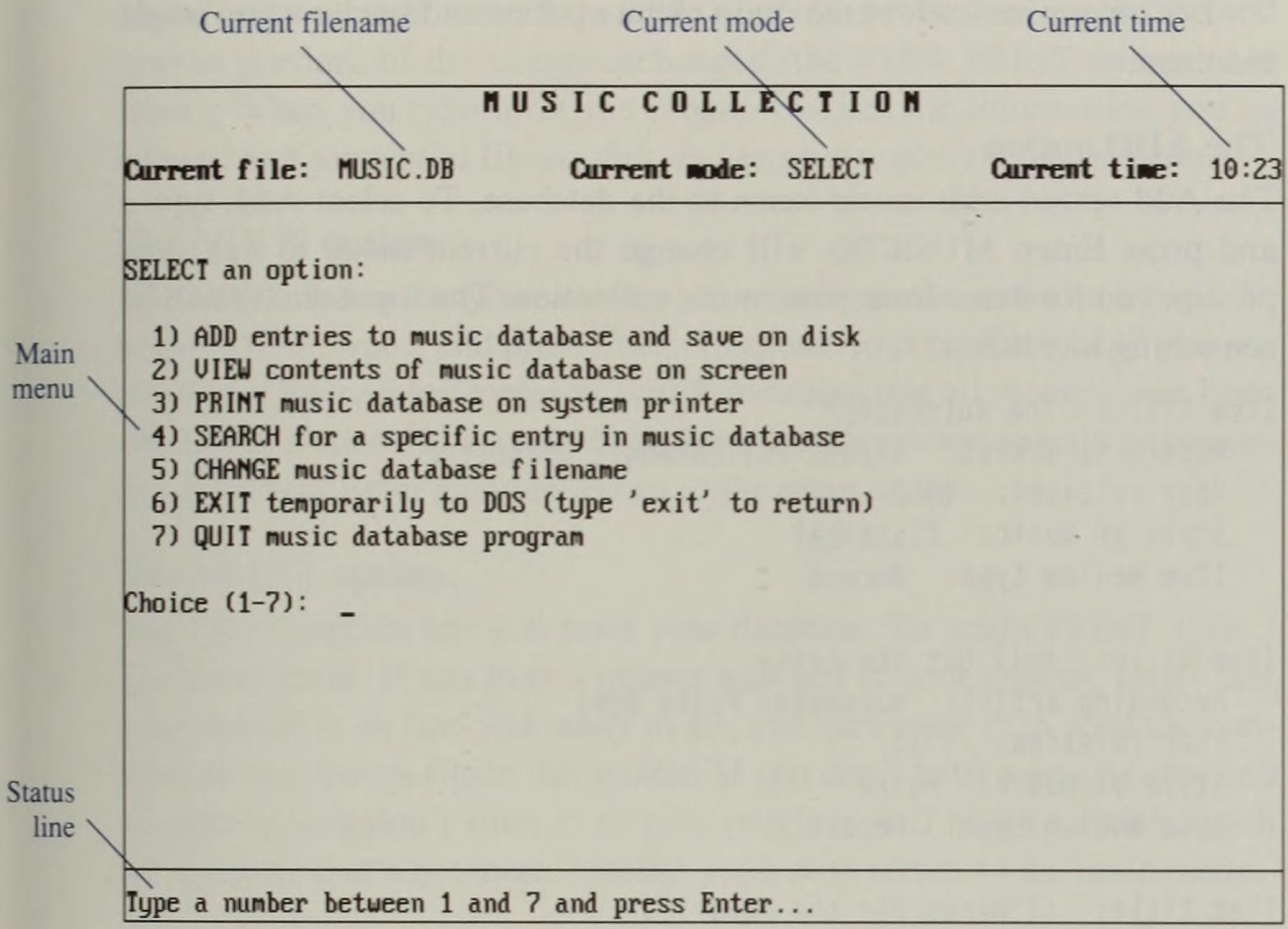
A *database* is a collection of individual records that have a format in common. A phone book, for example, is a database: It contains an alphabetic listing of name records, each of which contains address and telephone-number elements. A database always has a fixed structure; that is, each record contains the same *type* of information, though not necessarily the same *content*.

Databases are pretty common these days—turning up in both home and professional life. Here are a few everyday examples:

- Employee records
- Student records
- Available products
- A music collection
- A coin collection
- A film and video collection
- A parts catalog
- A library card catalog
- Important dates
- Inventory information
- Team statistics
- Financial transactions

A First Look at MUSICDB.BAS

Before you type in the MUSICDB.BAS program, take a few moments to see what the MUSIC collection program can do. When you start MUSICDB, you see the following screen:



```
                               MUSIC COLLECTION

Current file:  MUSIC.DB        Current mode:  SELECT        Current time:  10:23

SELECT an option:

  1) ADD entries to music database and save on disk
  2) VIEW contents of music database on screen
  3) PRINT music database on system printer
  4) SEARCH for a specific entry in music database
  5) CHANGE music database filename
  6) EXIT temporarily to DOS (type 'exit' to return)
  7) QUIT music database program

Choice (1-7):  _

Type a number between 1 and 7 and press Enter...
```

You'll see these elements on the music-collection screen:

- *Current filename* is the name of the music database file that is currently open (MUSIC.DB is the default).
- *Current mode* is the menu task that is currently active; this status indicator changes as different tasks are executed.
- *Current time* is the current time as read from the system clock (updated each time the main menu is displayed).
- *Main menu* contains the seven different tasks the program can accomplish; this is the base menu of the program and will reappear after each task has been carried out.
- *Status line* contains instructions or information about the menu selection that is currently active.

Let's examine each of the main menu options and see how you might use them.

The ADD option

The Add option adds music items to the database. To select Add, type *1* and press Enter. MUSICDB will change the current mode to *ADD* and prompt you for items from your music collection. The input dialog will go something like this:

```
Item title:  The Nutcracker
   Recording artist:  Vienna Philharmonic Orchestra
   Year released:  1982
   Style of music:  Classical
   Item medium type:  Record

Item title:  Roll Out the Lefse
   Recording artist:  Norwegian Polka Boys
   Year released:  1953
   Style of music:  Polka
   Item medium type:  Record

Item title:  (T)Hanks for the Memories
   Recording artist:  The Hanks
   Year released:  1989
   Style of music:  Rock
   Item medium type:  Compact disc

Item title:  Hurried Honeymoon
   Recording artist:  The Magees
   Year released:  1988
   Style of music:  Country
   Item medium type:  Record

Item title:  Der Ring des Nibelungen (Great Scenes)
   Recording artist:  Vienna Philharmonic Orchestra
   Year released:  1985
   Style of music:  Classical
   Item medium type:  Cassette

Item title:  END
```

If you enter enough items, the middle window scrolls, leaving the top and bottom portions of the screen unchanged (the VIEW PRINT statement at work). When you type *END*, the program writes the information you've entered to a sequential file on disk and returns you to the main menu.

The VIEW option

The VIEW option lets you examine your database one record at a time. To select VIEW, type *2* and press Enter. Continue to press Enter until you see the *End of file reached* message, which indicates that all records have been displayed. Notice that each record appears on the screen exactly as you entered it. Press Enter again to return to the main menu.

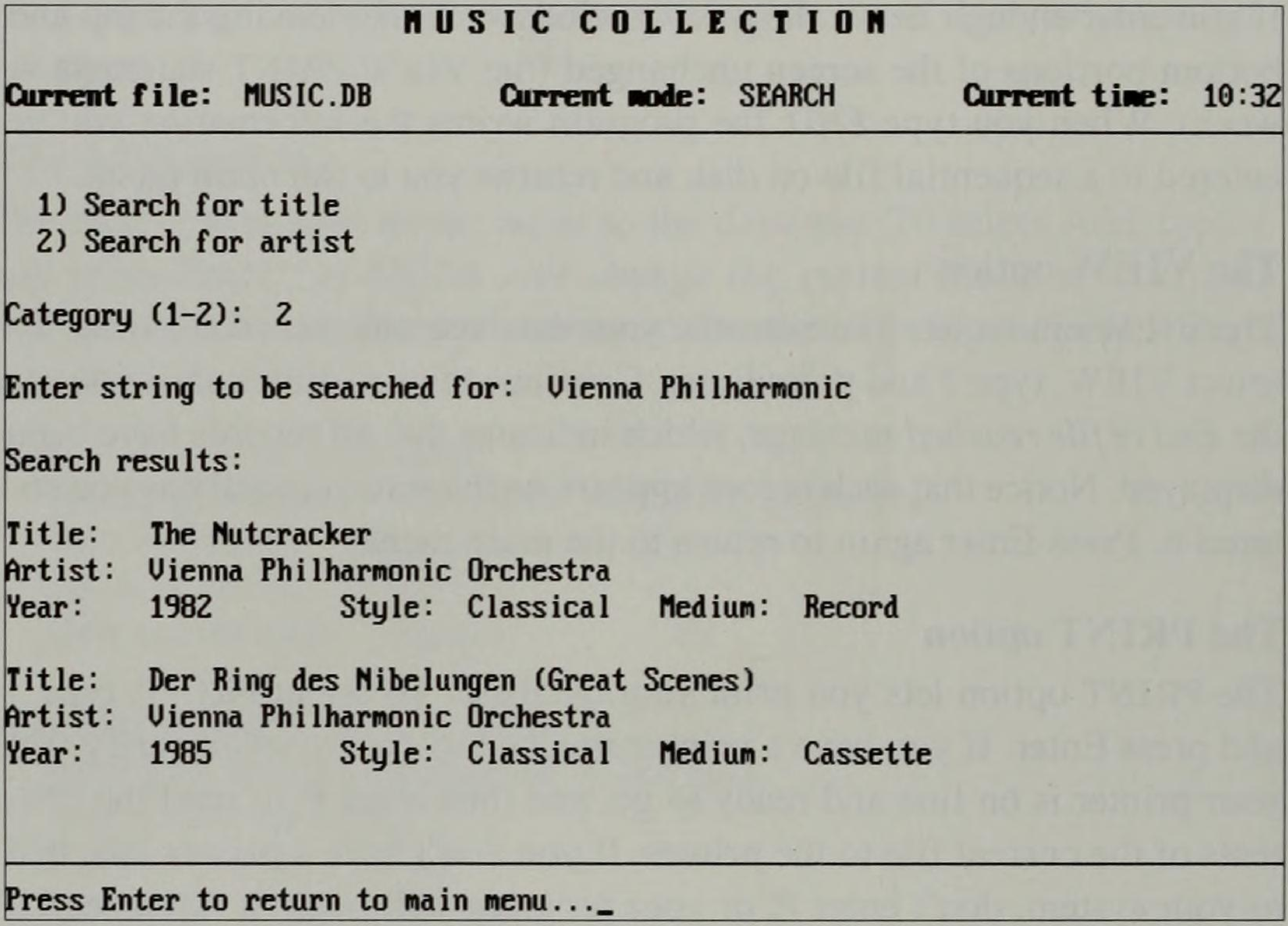
The PRINT option

The PRINT option lets you print your database. To select PRINT, type *3* and press Enter. If you have a printer attached to your system, verify that your printer is on line and ready to go, and then enter *P* to send the contents of the current file to the printer. If you don't have a printer attached to your system, don't enter *P*, or your program will begin a futile search for a printer that's not there. Instead, enter *R* to return to the main menu.

The SEARCH option

The SEARCH option lets you search for a specific title or artist in the database. To select the SEARCH option, type *4* and press Enter. When you are prompted for a category, you can type *1* to search for a title or *2* to search for an artist. The screen on the next page shows what will happen if you've entered the records from our example and then request an artist search for *Vienna Philharmonic.*

Two albums are displayed, with the item you were searching for—in this case, the artist's name—highlighted in green. After the search is complete, press Enter to return to the main menu.



```
                       M U S I C   C O L L E C T I O N

Current file:  MUSIC.DB         Current mode:  SEARCH         Current time:  10:32

  1) Search for title
  2) Search for artist

Category (1-2):  2

Enter string to be searched for:  Vienna Philharmonic

Search results:

Title:   The Nutcracker
Artist:  Vienna Philharmonic Orchestra
Year:    1982         Style:  Classical   Medium:  Record

Title:   Der Ring des Nibelungen (Great Scenes)
Artist:  Vienna Philharmonic Orchestra
Year:    1985         Style:  Classical   Medium:  Cassette

Press Enter to return to main menu..._
```

The CHANGE option

The CHANGE option lets you change the filename you are working with so that another music database (in the same format) can be examined or created. To select the CHANGE option, type *5* and press Enter. To refresh your memory about the files you already have, CHANGE displays the contents of the current directory before it prompts you for a new filename. You can include a drive and pathname in your new filename, but if you specify an invalid filename your program will terminate. If you press Enter without specifying a new filename, the default music database file, MUSIC.DB, will be selected.

NOTE: Specify only music database files (new or existing) with the CHANGE option. Do not specify document or program files from other applications (such as those with .BAS, .BAT, or .DOC extensions)—you might damage them.

The EXIT option

The EXIT option lets you exit to DOS temporarily so that you can carry out DOS commands or run small programs. (You won't be able to run large programs because QBasic and the MUSICDB program will still be in memory.) To select the EXIT option, type *6* and press Enter. After you've typed a few DOS commands, type *exit* and press Enter to return to the main menu of MUSICDB.

The QUIT option

The QUIT option lets you quit the MUSICDB program and return to QBasic. To select the QUIT option, type *7* and press Enter.

The MUSICDB.BAS Program Listing

Now that you've seen how the program works, take a look at the program listing to see how it uses what you've learned in this chapter and in previous chapters. MUSICDB.BAS (Figure 10-15) consists of one main program and seven subprograms. Although each subprogram handles a specific task, each is designed to be a general-purpose routine so that you can adapt it later to any database project that's of special interest to you.

We've highlighted items in the complete program listing to indicate where you need to modify the program if you eventually decide to adapt the database to track a different type of information. Take some time to examine the listing and read the explanatory comments. As you do, notice the parts of each subprogram that are independent of the type of data being worked with: choosing an item from the main menu; processing the menu item selection; opening and closing files; and reading, printing, and searching for items. Changing the program to fit another type of database—coin collection, employee record, or inventory—simply entails removing the music-specific fields and input prompts and adding appropriate fields and input prompts to the new database. The window dressing changes, but the internal structure remains the same.

Practice: Typing in the music-collection program

Type in the MUSICDB.BAS program and run it. Note that Figure 10-15 shows how MUSICDB.BAS looks when it is printed out—not how it looks inside QBasic. To type in this program, you need to use the New SUB command on the Edit menu to enter each subprogram in a separate window. For more information about this process, see Chapter 7.

```
' MUSICDB.BAS
' This is a simple database program that tracks a home music collection
'   with a sequential file and a number of general-purpose subprograms.

DECLARE SUB DisplayHeader ()            ' declare subprograms
DECLARE SUB GetMenuSelection (choice%)
DECLARE SUB AddRecords ()
DECLARE SUB ViewRecords ()
DECLARE SUB PrintRecords ()
DECLARE SUB Search ()
DECLARE SUB ChangeFilename ()

COMMON SHARED filename$, tmp$           ' declare global variables
filename$ = "MUSIC.DB"                  ' default database filename
OPEN filename$ FOR APPEND AS #1: CLOSE #1  ' ensure that file exists
tmp$ = "Year:    ####        Style:  \         \ Medium:  \           \"

DisplayHeader                           ' call sub to set up screen

DO
    GetMenuSelection choice%            ' call sub to get menu choice

    SELECT CASE choice%                 ' process menu choice
        CASE 1                             ' "1" means add to database
            LOCATE 3, 47: PRINT "ADD   "   '   change mode to ADD
            AddRecords                     '   call sub to add items
        CASE 2                             ' "2" means view database
            LOCATE 3, 47: PRINT "VIEW  "   '   change mode to VIEW
            ViewRecords                    '   call sub to view items
```

FIGURE 10-15. *(continued)*

MUSICDB.BAS: A general-purpose database program that uses a sequential file to track a home music collection. Modify the highlighted items when you adapt this program for another kind of database.

FIGURE 10-15. *continued*

```
        CASE 3                              ' "3" means print database
            LOCATE 3, 47: PRINT "PRINT "    '   change mode to PRINT
            PrintRecords                    '   call sub to print
        CASE 4                              ' "4" means search database
            LOCATE 3, 47: PRINT "SEARCH"    '   change mode to SEARCH
            Search                          '   call sub to search
        CASE 5                              ' "5" means change filename
            LOCATE 3, 47: PRINT "CHANGE"    '   change mode to CHANGE
            ChangeFilename                  '   call sub to change it
        CASE 6                              ' "6" means exit to DOS
            CLS                             '   clear screen
            SHELL                           '   exit to DOS shell
            DisplayHeader                   '   set up screen on return
        CASE 7                              ' "7" means quit program
            LOCATE 3, 47: PRINT "QUIT  "    '   change mode to QUIT
    END SELECT

LOOP UNTIL (choice% = 7)                 ' repeat loop until QUIT chosen

END

SUB AddRecords

' The AddRecords subprogram adds new music items to the database.

LOCATE 25, 1                         ' print message on status line
PRINT "Enter music data.  Type END for title to quit...";
VIEW PRINT 5 TO 23                   ' enable viewport (lines 5-23)
PRINT                                ' prompt for data
PRINT "Enter new music item information (without commas)"
PRINT

OPEN filename$ FOR APPEND AS #1      ' open database in append mode

' get records for file until user enters END for title

WHILE (UCASE$(title$) <> "END")
    INPUT "Item title:  ", title$    ' get item title
```

(continued)

FIGURE 10-15. *continued*

```
    IF (UCASE$(title$) <> "END") THEN
        INPUT "   Recording artist:  ", artist$   ' ...and other info
        INPUT "   Year released:  ", year%
        INPUT "   Style of music:  ", style$
        INPUT "   Item medium type:  ", medium$
        PRINT
                                   ' write record to database file
        WRITE #1, title$, artist$, year%, style$, medium$
    END IF
WEND

CLOSE #1                              ' close file when finished

END SUB

SUB ChangeFilename

' The ChangeFilename subprogram changes the name of the current
'   database file. If the new file does not exist, it is created.
'   If no filename is specified, the default value of MUSIC.DB is
'   assumed.  Note:  This subprogram does only minimal checking
'   for a valid DOS filename--if an invalid name is entered the
'   program will terminate.

LOCATE 25, 1: PRINT "Specify new music database filename...";
VIEW PRINT 5 TO 23                ' print message on status line

PRINT                             ' prompt for a new filename
PRINT "Use this option to create a new music database file or open";
PRINT " an existing one."
PRINT
PRINT "The current directory contains the following files:"
PRINT
FILES "*.*"                       ' display all files in the current
PRINT                             '   directory to help user
PRINT "What music-collection data file would you like to work with?"
PRINT "(Press Enter for default database file MUSIC.DB)"
```

(continued)

FIGURE 10-15. *continued*

```
PRINT
INPUT "Filename:  ", filename$     ' assign input to global variable

IF (filename$ = "") THEN           ' if no filename entered then
    filename$ = "MUSIC.DB"         '   set filename to MUSIC.DB
ELSE                               ' otherwise trim blank spaces off
    filename$ = LTRIM$(RTRIM$(UCASE$(filename$)))
END IF                             '   both ends of file and change to
                                   '   uppercase

OPEN filename$ FOR APPEND AS #1    ' open and close file to ensure it
CLOSE #1                           '   exists on disk (this avoids file
                                   '   error when opening in INPUT mode)
END SUB

SUB DisplayHeader

' The DisplayHeader subprogram displays the status information on the
'   first three lines of the screen and the two dividing lines that set
'   off program information window.

CLS                                       ' clear screen

COLOR 9                                   ' set color to light blue

PRINT "                          M U S I C  C O L L E C T I O N"
PRINT
PRINT "Current file:                   "; ' display status fields
PRINT "Current mode:                 ";
PRINT "Current time:"

PRINT STRING$(80, "-")                    ' print dividing lines
LOCATE 24, 1: PRINT STRING$(80, "-");     '   on lines 4 and 24

COLOR 7                                   ' set color to default white

END SUB
```

(continued)

FIGURE 10-15. *continued*

```
SUB GetMenuSelection (choice%)

' The GetMenuSelection subprogram gets a menu choice from the user
'   and returns it to the main program in the choice% variable.
'   The VIEW PRINT statement is used to enable and disable the
'   viewport area (lines 5-23). The information displayed here does
'   not disturb the data in lines 1 through 4 and 24 through 25.

choice% = 0                 ' initialize choice% to zero

VIEW PRINT                  ' disable viewport to update lines 3 and 25
LOCATE 3, 16: PRINT "                ": LOCATE 3, 16: PRINT filename$
LOCATE 3, 47: PRINT "SELECT"          ' set current mode to select
LOCATE 3, 76: PRINT LEFT$(TIME$, 5)    ' update current time
LOCATE 25, 1: PRINT "Type a number between 1 and 7 and press Enter...";
VIEW PRINT 5 TO 23          ' enable viewport (lines 5-23)
CLS 2                       ' clear viewport for choice prompts

PRINT                       ' prompt user for choice
PRINT "SELECT an option:"
PRINT
PRINT "  1) ADD entries to music database and save on disk"
PRINT "  2) VIEW contents of music database on screen"
PRINT "  3) PRINT music database on system printer"
PRINT "  4) SEARCH for a specific entry in music database"
PRINT "  5) CHANGE music database filename"
PRINT "  6) EXIT temporarily to DOS (type 'exit' to return)"
PRINT "  7) QUIT music database program"
PRINT
                            ' choice must be integer between 1 and 7
DO WHILE (choice% < 1) OR (choice% > 7)
    INPUT "Choice (1-7):  ", choice%
LOOP

CLS 2                       ' clear viewport for upcoming choice
VIEW PRINT                  ' disable viewport to clear status line
LOCATE 25, 1: PRINT STRING$(80, " ");  ' print a blank line

END SUB
```

(continued)

FIGURE 10-15. *continued*

```
SUB PrintRecords

' The PrintRecords subprogram sends the entire contents of the current
'   database file to the printer.

VIEW PRINT 5 TO 23                    ' enable viewport (lines 5-23)
PRINT                                 ' display introductory message
PRINT "This option sends the contents of "; filename$;
PRINT " to your printer."

VIEW PRINT                            ' disable viewport so status
LOCATE 25, 1                          '   line can be updated
INPUT ; "Enter P to print or R to return to main menu:  ", reply$
VIEW PRINT 5 TO 23                    ' enable viewport (lines 5-23)
                                      ' if user wants to print (P or p)
IF (reply$ = "P") OR (reply$ = "p") THEN
    OPEN filename$ FOR INPUT AS #1    ' open the music database file
                                      ' send header to printer
    LPRINT "------------------- Music Collection ------------------"
    LPRINT
    LPRINT "Date printed:  "; DATE$  ' print current date
    LPRINT "Filename:      "; filename$  ' print current filename
    LPRINT
    LPRINT "Collection contents:"
    LPRINT
                                      ' until file contents exhausted
    DO WHILE (NOT EOF(1))             '   read a record from file
        INPUT #1, title$, artist$, year%, style$, medium$

        LPRINT "Title:   "; title$    '  print each field of the record
        LPRINT "Artist:  "; artist$
        LPRINT "Year:    "; year%
        LPRINT "Style:   "; style$
        LPRINT "Medium:  "; medium$
        LPRINT
    LOOP
```

(continued)

FIGURE 10-15. *continued*

```
    LPRINT CHR$(12)               ' send formfeed character to printer
    CLOSE #1                      ' close file
END IF

END SUB

SUB Search

' The Search subprogram searches the entire database file for records
'   matching a search string entered by the user.  Search currently
'   supports searches for title and artist fields--additional topics
'   can be included by adding extra CASE statements.

num% = 0                              ' initialize category variable
found% = 0                            ' initialize "record found" flag

LOCATE 25, 1                          ' update status line
PRINT "Enter search category and content...";
VIEW PRINT 5 TO 23                    ' enable viewport (lines 5-23)

PRINT
PRINT "Select a search category:"     ' prompt for search topic
PRINT
PRINT "  1) Search for title"
PRINT "  2) Search for artist"
' add prompts for additional search categories here...
PRINT

DO WHILE (num% < 1) OR (num% > 2)     ' get number associated with
    INPUT "Category (1-2):  ", num%   '   search topic
LOOP

PRINT                                 ' get search string
INPUT "Enter string to be searched for:  ", searchStr$
PRINT
PRINT "Search results:"               ' display search results
PRINT
```

(continued)

FIGURE 10-15. *continued*

```
OPEN filename$ FOR INPUT AS #1        ' open database file

DO WHILE (NOT EOF(1))                 ' read records from file
    INPUT #1, title$, artist$, year%, style$, medium$

    SELECT CASE num% ' use num% to compare correct record field...
    CASE 1   ' if num% = 1, determine if search string in title field
        IF INSTR(UCASE$(title$), UCASE$(searchStr$)) THEN
            found% = -1               ' if so, set found flag to true
            COLOR 2: PRINT "Title:   "; title$: COLOR 7
            PRINT "Artist:  "; artist$
            PRINT USING tmp$; year%; style$; medium$
            PRINT    ' display record fields with title field in green
         END IF
    CASE 2   ' if num% = 2, determine if search string in artist field
        IF INSTR(UCASE$(artist$), UCASE$(searchStr$)) THEN
            found% = -1               ' if so, set found flag to true
            PRINT "Title:   "; title$
            COLOR 2: PRINT "Artist:  "; artist$: COLOR 7
            PRINT USING tmp$; year%; style$; medium$
            PRINT    ' display record fields with artist field in green
         END IF
    ' add CASE statements for additional categories here...
    END SELECT
LOOP

CLOSE #1                              ' close file
IF (NOT found%) THEN                  ' if file not found display
    COLOR 2: PRINT searchStr$;        '   "not found" message
    COLOR 7: PRINT " not found in "; filename$; " database"
END IF

VIEW PRINT           ' disable viewport and update status line
LOCATE 25, 1: INPUT ; "Press Enter to return to main menu...", dummy$

END SUB
```

(continued)

FIGURE 10-15. *continued*

```
SUB ViewRecords

' The ViewRecords subprogram displays each record in the database on
'   the screen one at a time.

LOCATE 25, 1                           ' update status line
PRINT "Press Enter to continue...";

VIEW PRINT 5 TO 23                     ' enable viewport (lines 5-23)
PRINT                                  ' display opening message
PRINT "This option lets you view your music collection ";
PRINT "one record at a time."
PRINT

OPEN filename$ FOR INPUT AS #1         ' open database file

DO WHILE (NOT EOF(1))                  ' get record from file
    INPUT #1, title$, artist$, year%, style$, medium$

    PRINT "Title:   "; title$          ' display each field on screen
    PRINT "Artist:  "; artist$
    PRINT USING tmp$; year%; style$; medium$

    INPUT "", dummy$                   ' pause after each record
LOOP

CLOSE #1                               ' close file
PRINT "** End of file reached **"      ' display EOF message
INPUT "", dummy$                       ' pause before returning
                                       '   to main program
END SUB
```

SUMMARY

In this chapter you've started to put your skills together in a really useful way. You've continued to work with large amounts of data by learning how to read and write sequential files. You've also learned about the parts that DOS and DOS commands play in QBasic programming, and you've practiced using a number of DOS commands in useful ways. In addition, you've worked with a music-collection database program that combines the string-manipulation and user-interface techniques you developed in earlier chapters with the sequential-file skills you've learned in this chapter. The result: A general-purpose database program you can use to track information *you* are interested in.

QUESTIONS AND EXERCISES

1. What is the difference between opening a file for OUTPUT and opening a file for APPEND?
2. Which of the following statements encloses data items in quotation marks when it sends them to a sequential file?

   a. INPUT#

   b. PRINT# USING

   c. PRINT#

   d. WRITE#
3. True or False: The filename specified in an OPEN statement must be in uppercase letters.
4. When is the LINE INPUT# statement considered more useful than the INPUT# statement?
5. What is wrong with the following QBasic statement?

   ```
   SHELL COPY TEST.TXT TEST2.TXT
   ```
6. What do the following program lines display?

   ```
   searchStr$ = "Bea"
   artist$ = "The Beatles"
   IF INSTR(UCASE$(artist$), UCASE$(searchStr$)) THEN
       PRINT "Search string found!"
   ELSE
       PRINT "Search string not found"
   END IF
   ```

7. Write a program that prompts the user for a list of cities and stores the information in a sequential file. Design the program so that the user can view the contents of the file after it has been created.
8. Write a program that prompts the user for a list of names and addresses, stores them in a sequential file, and then sorts the records in a file alphabetically by name. Hint: The easiest way to solve this problem is to use an array to store the records and sort them. You might want to use the Shell Sort, which you learned about in Chapter 9.

CHAPTER 11

Working with Graphics and Sound

Now that you've learned the basics of QBasic, you can enhance your programs by adding graphics and sound to them. In this chapter you'll learn what kind of graphics your computer can create, and you'll learn how to use sound to jazz up your programs.

INTRODUCTION TO GRAPHICS PROGRAMMING

Your hardware determines the type of graphics you can create, so it's important to know what kind of video equipment is installed in your computer.

What Is the Video Equipment?

On an IBM PC or compatible, the video equipment consists of two pieces:

- The *video adapter,* which is an electronic card installed inside your machine. (Note: On some computers, this equipment is already built into the computer—it does not come on a separate card.)
- The *monitor* (sometimes called the *video display*), which is the TV-like box that usually sits on top of your computer.

Some video equipment displays both graphics and color; some can display only graphics; some can't display graphics or color at all. And of the video equipment that displays both graphics and color, you have a wide variety of choices. Generally, the less expensive your equipment, the fewer colors and the ''grainier'' images you'll have. More-expensive video equipment usually displays images with many colors and a smooth, professional appearance.

Video adapters

Table 11-1 lists the video adapters available for the IBM PC, IBM PC/XT, IBM PC/AT, and compatible computers. Depending on the computer model, IBM PS/1 and PS/2 computers come with MCGA, VGA, Super VGA, or XGA graphics adapters.

Adapter	*Multicolor display*	*Graphics*
MDA (monochrome display adapter)	No	No
HGC (Hercules graphics card)*	No	Yes
CGA (color/graphics adapter)	Yes	Yes
EGA (enhanced graphics adapter)	Yes	Yes
MCGA (multicolor gate array)	Yes	Yes
VGA (video graphics array)	Yes †	Yes

* The Hercules graphics card is not manufactured by IBM. This type of card is often referred to as a *monographics adapter* or a *monochrome graphics adapter*.

† To receive multicolor output, you must have a color VGA monitor.

TABLE 11-1.
Video adapters for the IBM PC family of computers and compatibles.

You can install several makes of video adapters, but most are compatible with the video adapters listed in Table 11-1.

If your computer has an MDA adapter, you won't be able to do some of the graphics practices in this chapter. If you aren't sure what kind of adapter you have, read on. In a while, you'll type in and run a program that will tell you whether you can do the practices in this chapter.

Monitors

Like video adapters, monitors come in a wide variety of types (with, of course, a wide variety of prices). Although a certain amount of mixing and matching is allowed—in other words, certain video adapters work with several different types of monitors—most computer dealers generally match the video adapter you choose to the appropriate monitor.

Text Mode *vs* Graphics Mode

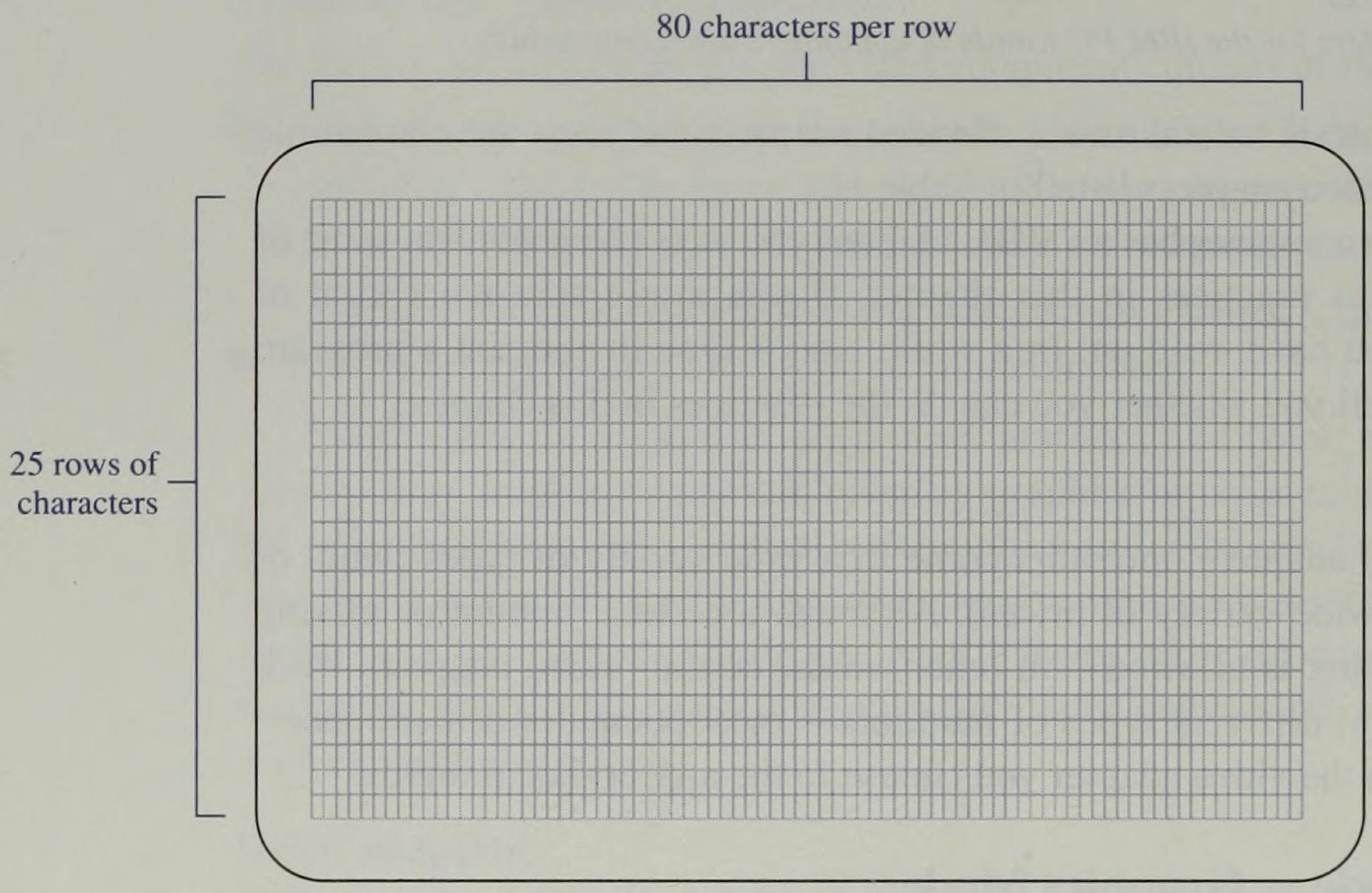
You can set any video adapter that can display graphics in either *text mode* or *graphics mode*.

- In text mode, your video adapter can display alphanumeric characters—letters, numbers, and punctuation marks—but cannot display shapes (such as lines, boxes, and circles).
- In graphics mode, your video adapter can still display alphanumeric characters, but it can also display non-alphanumeric shapes such as circles and polygons.

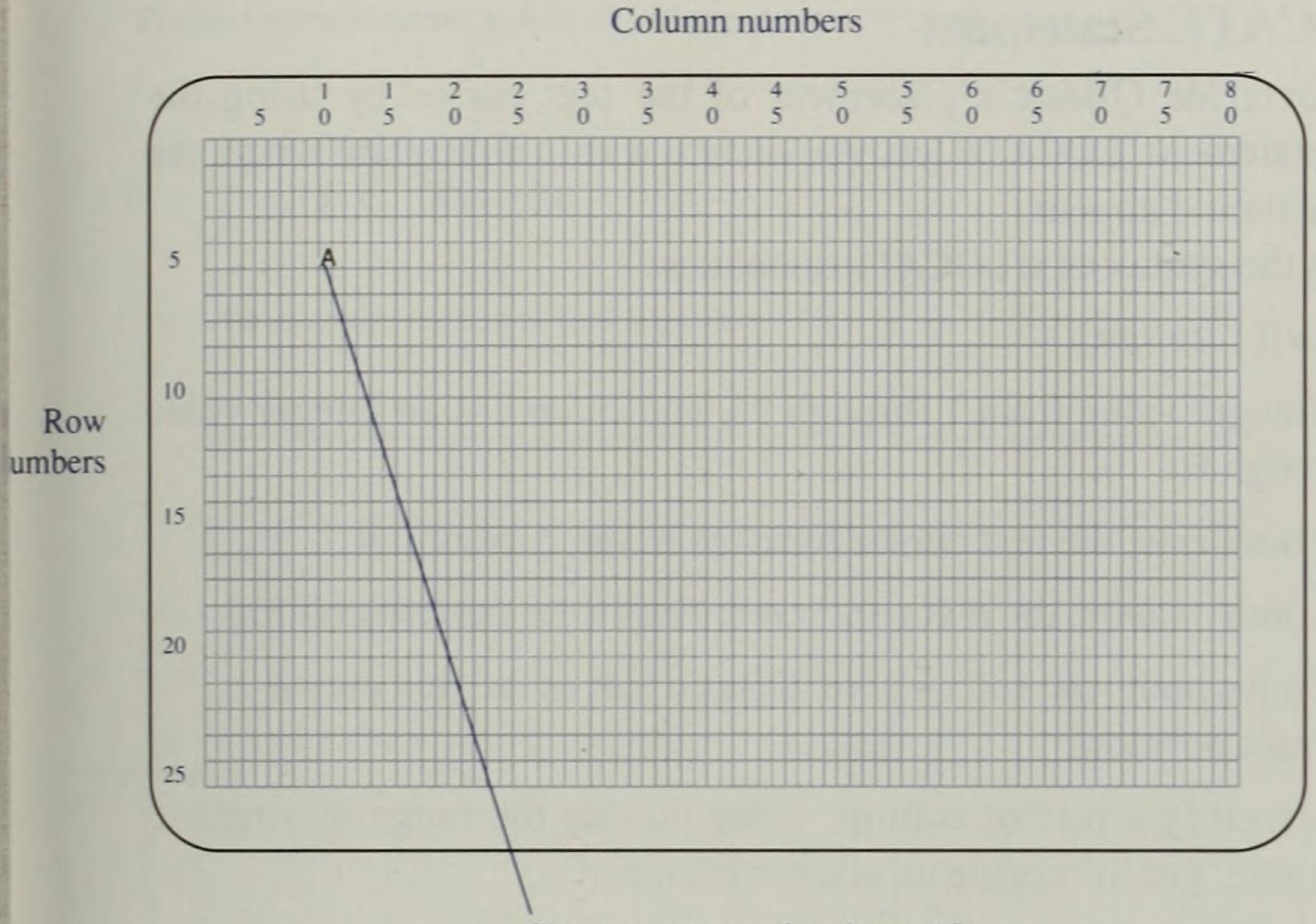
  We'll look at these two modes in detail.

TEXT MODE

When you start up QBasic, your video adapter displays both the View window and the output screen in *text mode*: 25 rows and 80 characters per row.



Each row and column in the output screen is numbered, giving each "box" unique row and column coordinates, called *screen coordinates*. Each box can hold only one alphanumeric character.



Character at row 5, column 10

The Text Cursor

You'll remember that as you type a program in the View window, the blinking cursor indicates where the next character you type will appear. On the output screen, QBasic uses its own cursor, called the *text cursor,* to determine where characters will appear. Unlike the blinking cursor in the View window, the text cursor is usually invisible.

The CLS statement that clears the output screen serves another purpose, too: CLS sets the text cursor to row 1, column 1 of the output screen. If QBasic then executes a PRINT statement, the PRINT statement prints its message starting at row 1, column 1. Assuming that the PRINT statement doesn't end in a semicolon, QBasic then moves the text cursor down to the first column of the next row (row 2, column 1).

The LOCATE Statement

You can override QBasic's placement of the text cursor by using the LOCATE statement. LOCATE lets you dictate within a program where the text cursor should appear.

Here's the syntax of a LOCATE statement:

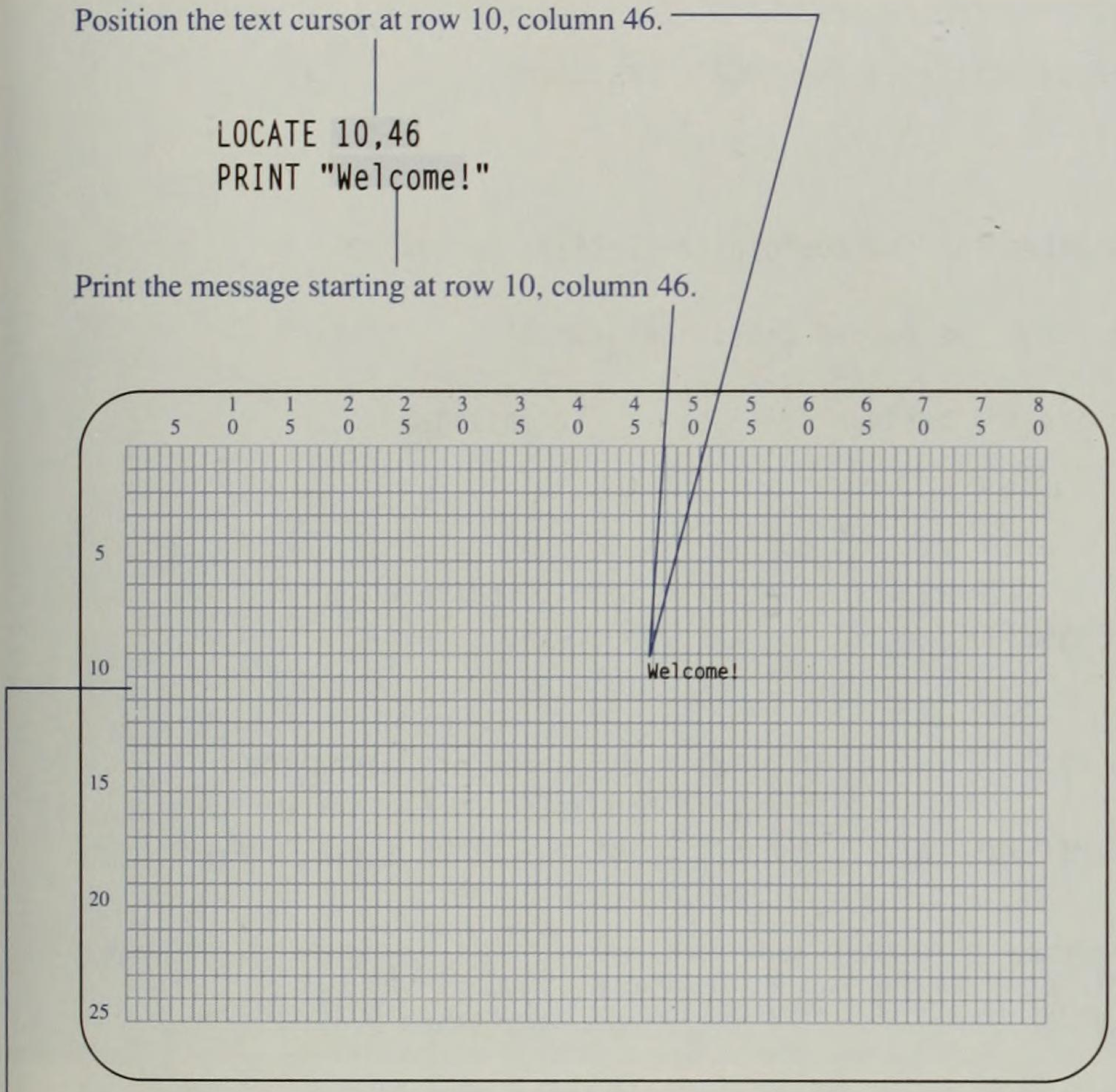
```
LOCATE [row][, column]
```

row is an integer value from 1 through 25, and *column* is an integer value from 1 through 80.

- If you omit *row*, the text cursor remains in the current row.
- If you omit *column*, the text cursor remains in the current column.
- If you omit both *row* and *column*, QBasic leaves the text cursor in its current position.
- If you specify a row or column value outside the range of your screen size, you'll receive an error message.

NOTE: As you'll learn shortly, you can instruct your video adapter to display only 40 columns of text across the screen. In this case, column *must be a value no greater than 40.*

In the following example, the LOCATE statement tells QBasic to put the cursor at row 10, column 46. When QBasic executes the subsequent PRINT statement, it prints the message *Welcome!* beginning at row 10, column 46. After printing that message, QBasic places the text cursor at row 11, column 1, and will print the message of the next PRINT statement beginning at that point unless your program contains another LOCATE statement that changes the position of the text cursor.



Position the text cursor at row 10, column 46.

```
LOCATE 10,46
PRINT "Welcome!"
```

Print the message starting at row 10, column 46.

5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
5
10
15
20
25
Welcome!

QBasic moves the text cursor to row 11, column 1, after the message is printed.

Practice: Working with the LOCATE statement

1. Type in the LOCATE-1.BAS program (Figure 11-1 on the next page).

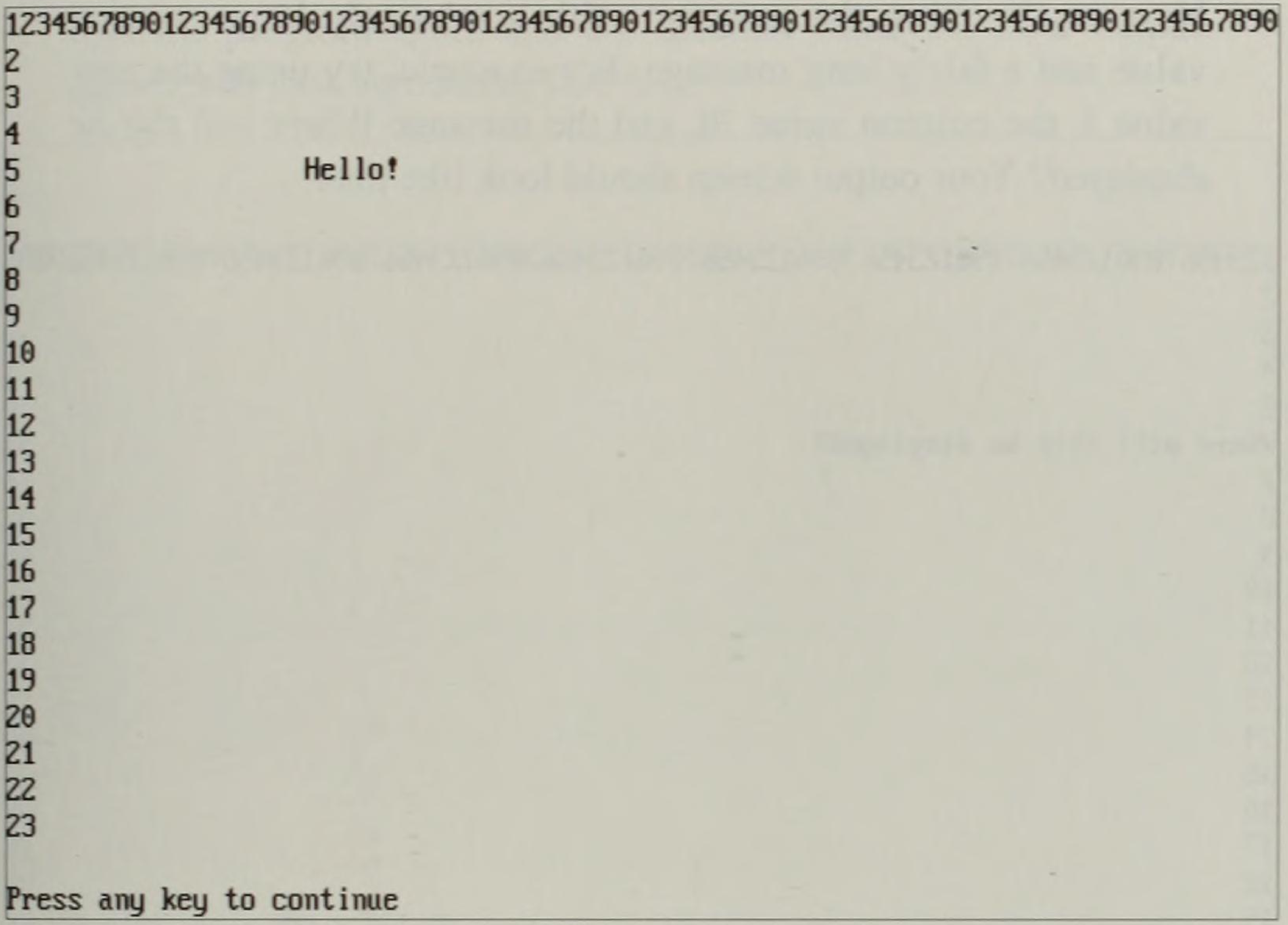
```
' LOCATE-1.BAS
' This program demonstrates the LOCATE statement.

CLS

INPUT "Please enter the row coordinate (1-24):  ", rowNum%
PRINT
INPUT "Please enter the column coordinate (1-80):  ", colNum%
PRINT
INPUT "Please enter a message to display:  ", message$

CLS

FOR i% = 0 TO 70 STEP 10      ' print column coordinates across
    PRINT "1234567890";       '   the top of the screen
NEXT i%

FOR i% = 2 TO 23              ' print the row numbers along the
    LOCATE i%, 1              '   left-hand side of the screen
    PRINT LTRIM$(STR$(i%))    ' don't print the blank space
NEXT i%                       '   before each number

LOCATE rowNum%, colNum%       ' print the message at the user-
PRINT message$                '   supplied coordinates
```

FIGURE 11-1.
LOCATE-1.BAS: A program demonstrating the LOCATE statement.

2. Run the program and enter row and column values that are greater than 2 (you'll see why in a moment). After you run the program, your output screen will look something like the screen shown at the top of the next page.

   Notice that the program prints numbers across row 1 and down columns 1 and 2. The program does this so that you can easily verify that your message begins to print at the coordinates whose values you typed in for the LOCATE statement.

   This example uses a row value of *5*, a column value of *20*, and the message *Hello!* Run the program a few times so that you can fully understand how the LOCATE statement works.



```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
2
3
4
5                  Hello!
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Press any key to continue
```

The STR$ Function

The STR$ function converts a numeric value to a string of numerals. The syntax of the STR$ function is as follows:

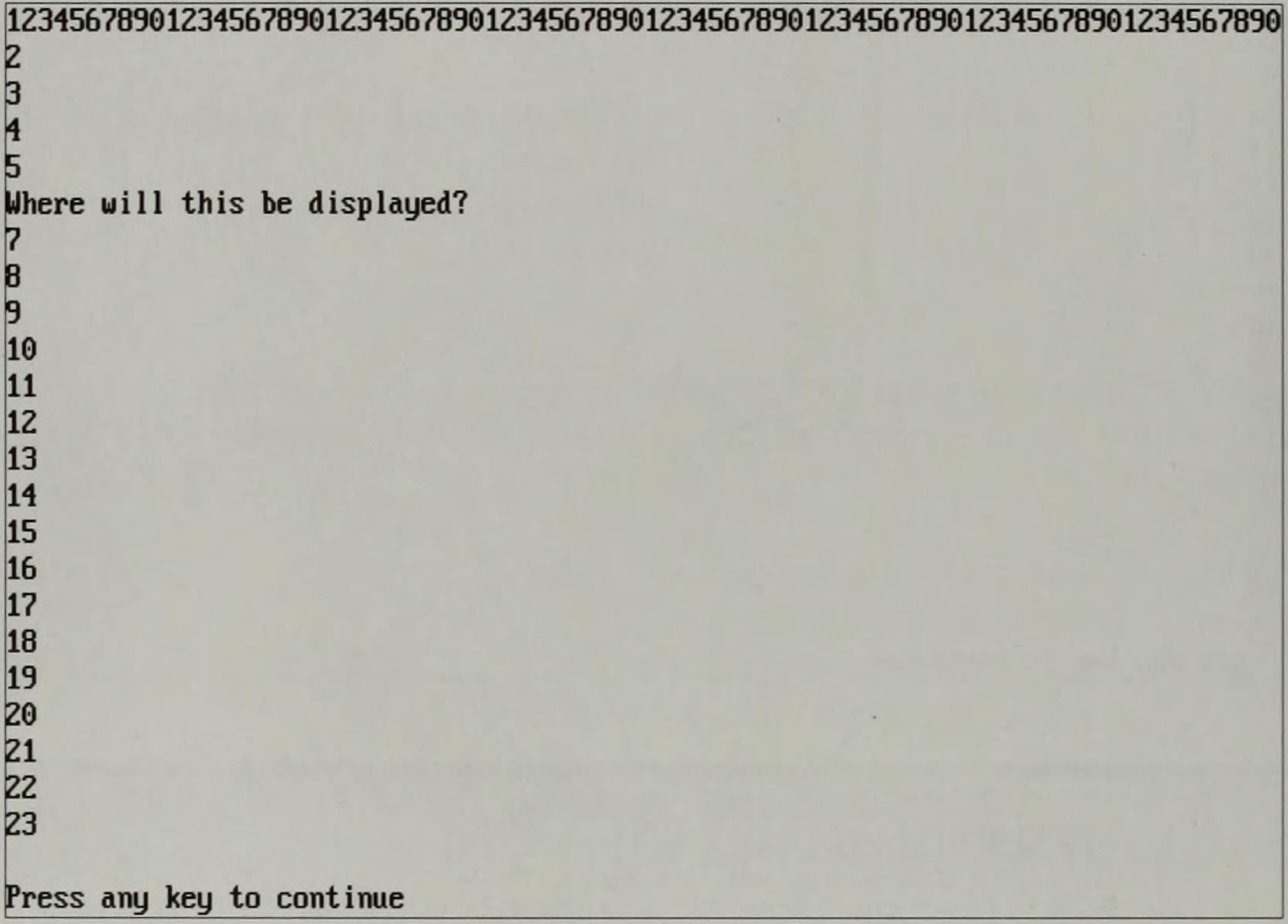
`STR$(value)`

value is any numeric expression.

You can use STR$ together with the LTRIM$ function to get rid of the space that appears to the left of non-negative numbers. This comes in handy when you position a number using the LOCATE statement. For example, the following statements display the number *12.34* with no leading space:

```
number! = 12.34
PRINT LTRIM$(STR$(number!))
```

3. Next, try running the program with a relatively large column value and a fairly long message. For example, try using the row value *5*, the column value *70*, and the message *Where will this be displayed?* Your output screen should look like this:

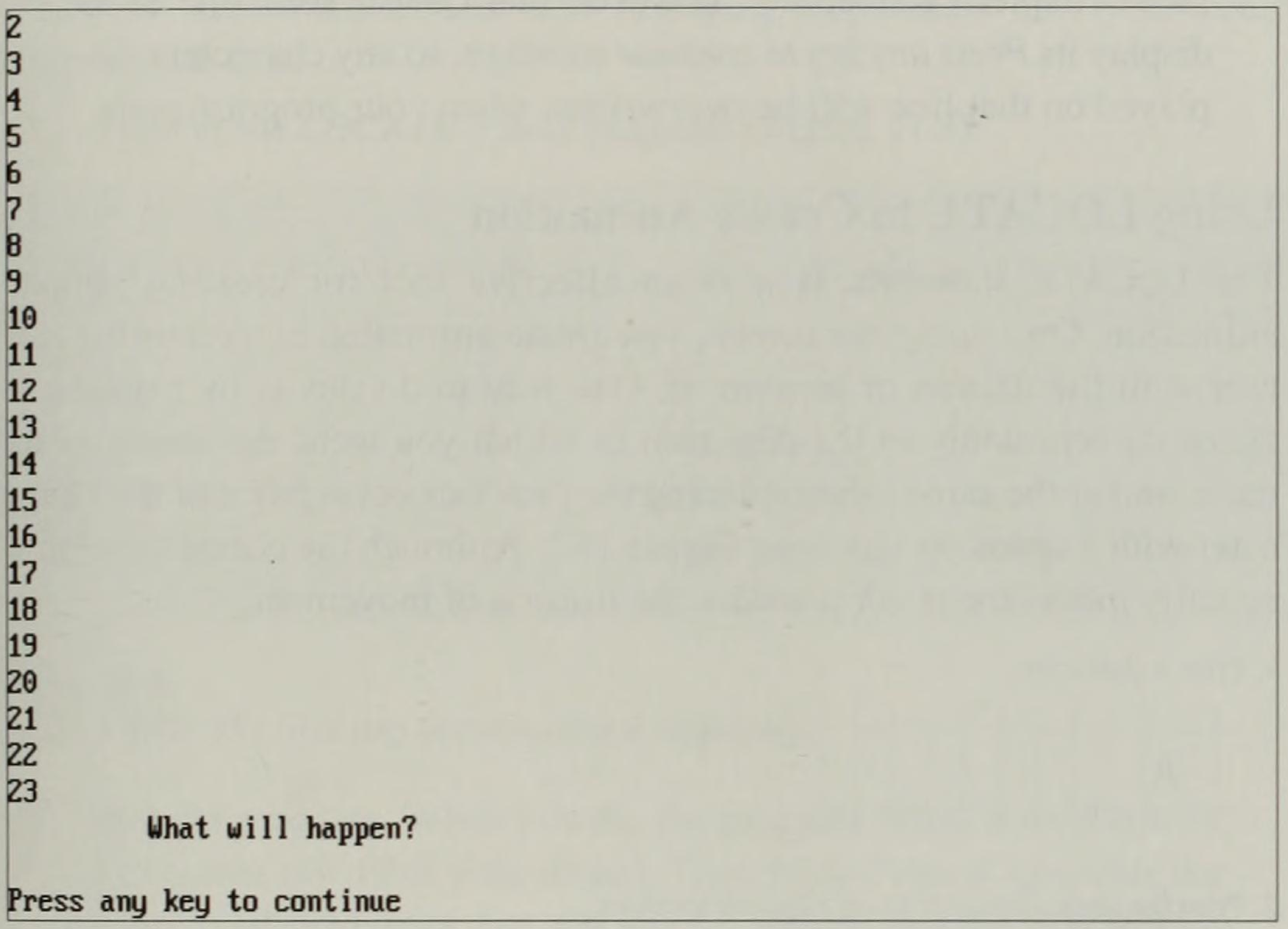


```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
2
3
4
5
Where will this be displayed?
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Press any key to continue
```

Notice that QBasic displayed the message starting at row 6, column 1, even though you asked QBasic to display it on row 5. Because the message exceeded the 80-character boundary, QBasic knew it couldn't fit the entire message on row 5, so it started at the beginning of the next row. Keep this in mind as you use the LOCATE statement, and plan accordingly.

You may be curious about one thing. The program asks you for a row value from 1 through 23 and numbers the rows on the screen only up through row 23. However, earlier we explained that the row value can be 1 through 25. Why did we limit the program to 23 rows?

4. Run the program again and use the row value *24*. Your output screen will look something like this:



```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
          What will happen?

Press any key to continue
```

After printing the message, QBasic moved the cursor down to the first column of the next row. When it moved the cursor down, QBasic had to shift the contents of the screen up one row so that the next row would be displayed. Doing this, as you can see, caused the topmost row to be pushed up off the screen.

You can solve this problem by adding a semicolon to the end of the PRINT statement to keep the cursor on the same line. The following statements, for example, display the message *This is line 24* on row 24 without shifting the contents of the screen:

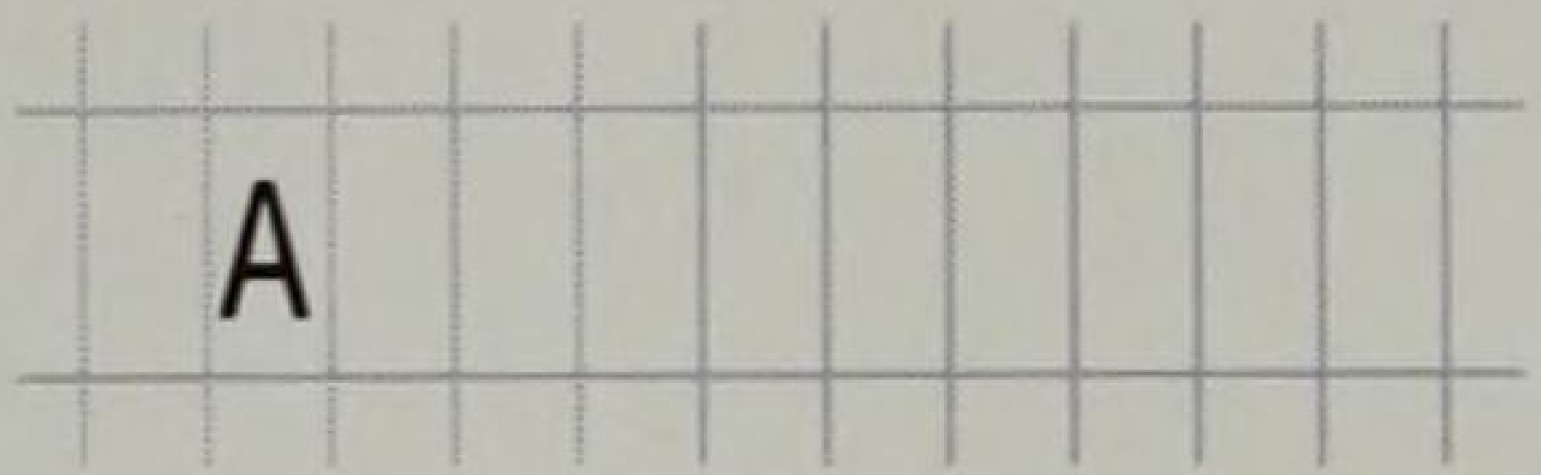
```
LOCATE 24, 1
PRINT "This is line 24";
```

You can use this same technique to place text on line 25. (You might remember this from the MUSICDB.BAS program in the previous chapter.) Remember, however, that QBasic uses line 25 to display its *Press any key to continue* message, so any characters displayed on that line will be overwritten when your program ends.

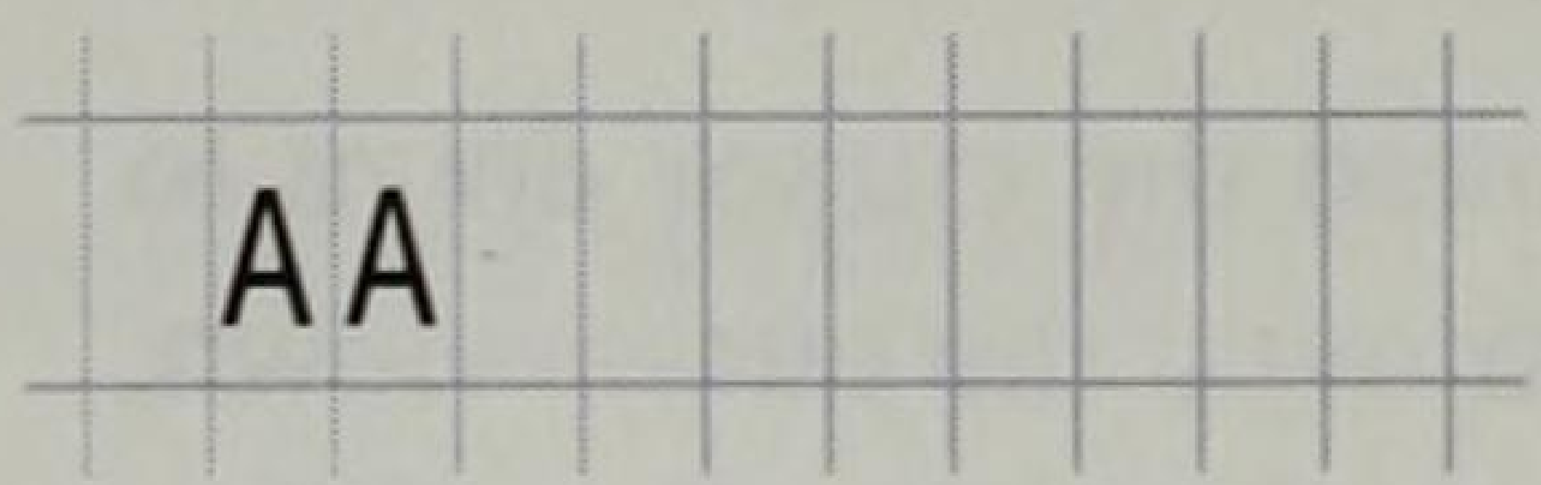
Using LOCATE to Create Animation

The LOCATE statement is also an effective tool for creating simple animation. On a computer screen, you create animation by presenting the user with the *illusion* of movement. One way to do this is by printing a character repeatedly in the direction in which you want the character to move and at the same time replacing the previous occurrence of the character with a space, as shown in Figure 11-2. Although the characters don't actually move, the result provides the illusion of movement.

1. Print a character.



2. Print the same character in an adjacent location.



3. Print a space where the first character was.

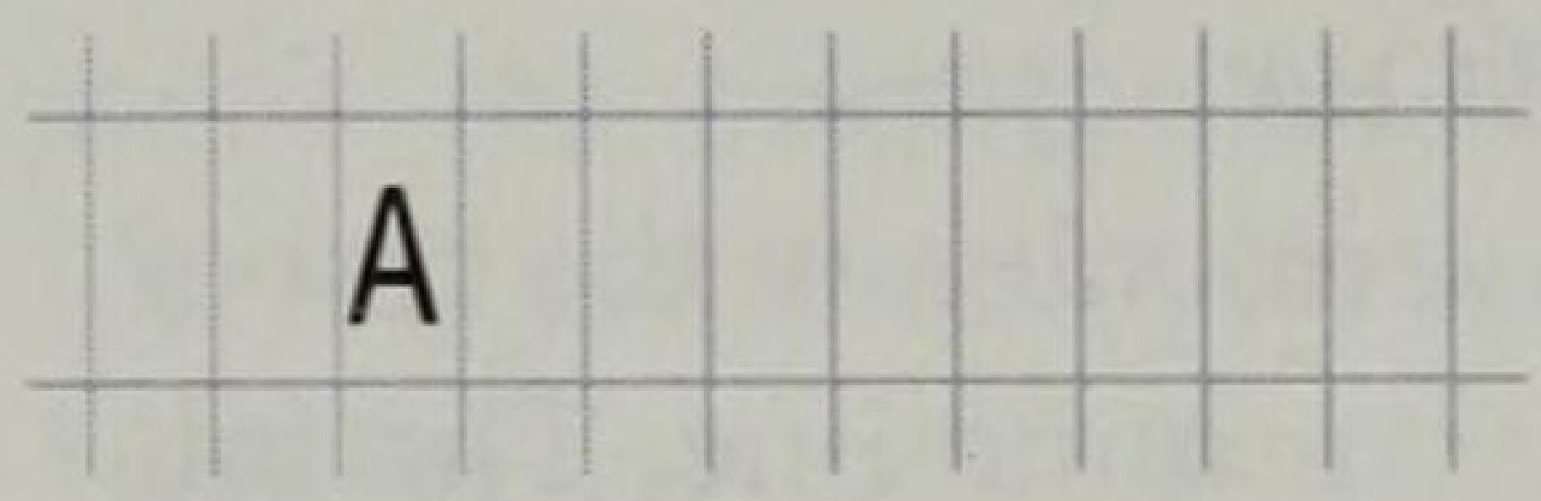


FIGURE 11-2.
Animating a character on the screen.

Practice: Working with text-based animation

Say you want to write a program that causes the letter X to "move" across the screen. The following steps show how you would accomplish this.

1. Type in the LOCATE-2.BAS program (Figure 11-3).

```
' LOCATE-2.BAS
' This program demonstrates the first step in creating animation.

CLS

FOR i% = 1 to 80
   LOCATE 10, i%
   PRINT "X"
NEXT i%
```

FIGURE 11-3.
LOCATE-2.BAS: The first step in animating a character.

2. Run the program. When you do, the program prints a solid row of X's across row 10 of your screen. This in itself doesn't provide the illusion of movement, but you have accomplished the first step—you've caused the letter X to "move" across the screen. Because QBasic executes the program quickly, it prints the row of X's almost instantaneously. Use the techniques you learned in Chapter 6 to put a delay loop in the program to slow the rate at which QBasic prints the X's. (The next program includes a delay loop, but try to create one yourself before we move on.)

   Next you need to create the illusion of making a *single* letter X move across the screen, even though the program will actually be printing 80 separate X's. To do this you have your program print an X, print a second X immediately to the right of the first, and then erase the first X by overwriting it with a space.

3. Type in the LOCATE-3.BAS program (Figure 11-4 on the next page) and run it.

```
' LOCATE-3.BAS
' This program demonstrates simple animation.

CLS

LOCATE 10, 1            ' position the first X
PRINT "X"
FOR i% = 2 to 80        ' loop through the rest
    LOCATE 10, i%
    PRINT "X"           ' print an X next to the previous X
    LOCATE 10, i% - 1
    PRINT " "           ' "erase" the previous X

    FOR j% = 1 TO 300   ' delay loop
    NEXT j%

NEXT i%
```

FIGURE 11-4.
LOCATE-3.BAS: A program that animates a character.

Notice that the first X is printed before the loop begins. The reason for this lies in how the X's are erased. The LOCATE values for printing a space are *10, i% – 1*. If the loop started with the value 1, the first time the program tried to print a space it would try to print it at row 10, column 0. Because 0 is not a valid value, QBasic would stop and display an error message.

4. Run the program. When you do, you should see the letter X ''move'' right across the screen! If you'd like, stop here and experiment with the program. Including such animation in your own programs can really get the user's attention!

Vertical animation

Now that you've learned the basics of text-based animation, let's look at how you can use LOCATE to accomplish some fancier animation: dropping words from the sky!

Practice: Working with FALLING.BAS

Type in FALLING.BAS (Figure 11-5) and run the program.

```
' FALLING.BAS
' This program demonstrates vertical animation.

CLS

FOR i% = 1 TO 4

    READ column%, word$

    FOR row% = 2 TO 18
        LOCATE row%, column%
        PRINT word$
        LOCATE row% - 1, column%
        PRINT SPACE$(LEN(word$))
        SOUND (2400 / row%), 1
    NEXT row%

NEXT i%

DATA 28, "The", 32, "sky", 36, "is", 39, "falling!!"
```

FIGURE 11-5.
FALLING.BAS: A program demonstrating vertical animation.

Instead of printing only a single letter, as in the previous example, this program prints entire words. And to ''erase'' the previous word, this program uses a string of spaces to cover the entire word at the same time.

In addition, this program uses sound to reinforce the ''falling'' illusion of each word. Notice how the SOUND statement uses the current value of *row%* as a divisor of the value *2400*. As the value of *row%* increases, the result of the division operation decreases and the resulting pitch of the tone becomes lower. (You'll learn more about the SOUND statement later in this chapter.)

GRAPHICS MODE

As you've seen, text mode can be an interesting and relatively straightforward mode for presenting information on the screen. You can add both text-based ''graphics'' and simple animation to your programs without too much programming effort.

The downside of text-based programming is its limitations. Even the smallest character on the screen, such as a period, takes up an entire character's ''box,'' meaning that smooth, well-defined graphics are difficult if not impossible to create. Also, text-based animation often seems jumpy because the item you animate has to jump the width of a character ''box'' each time your program ''moves'' it.

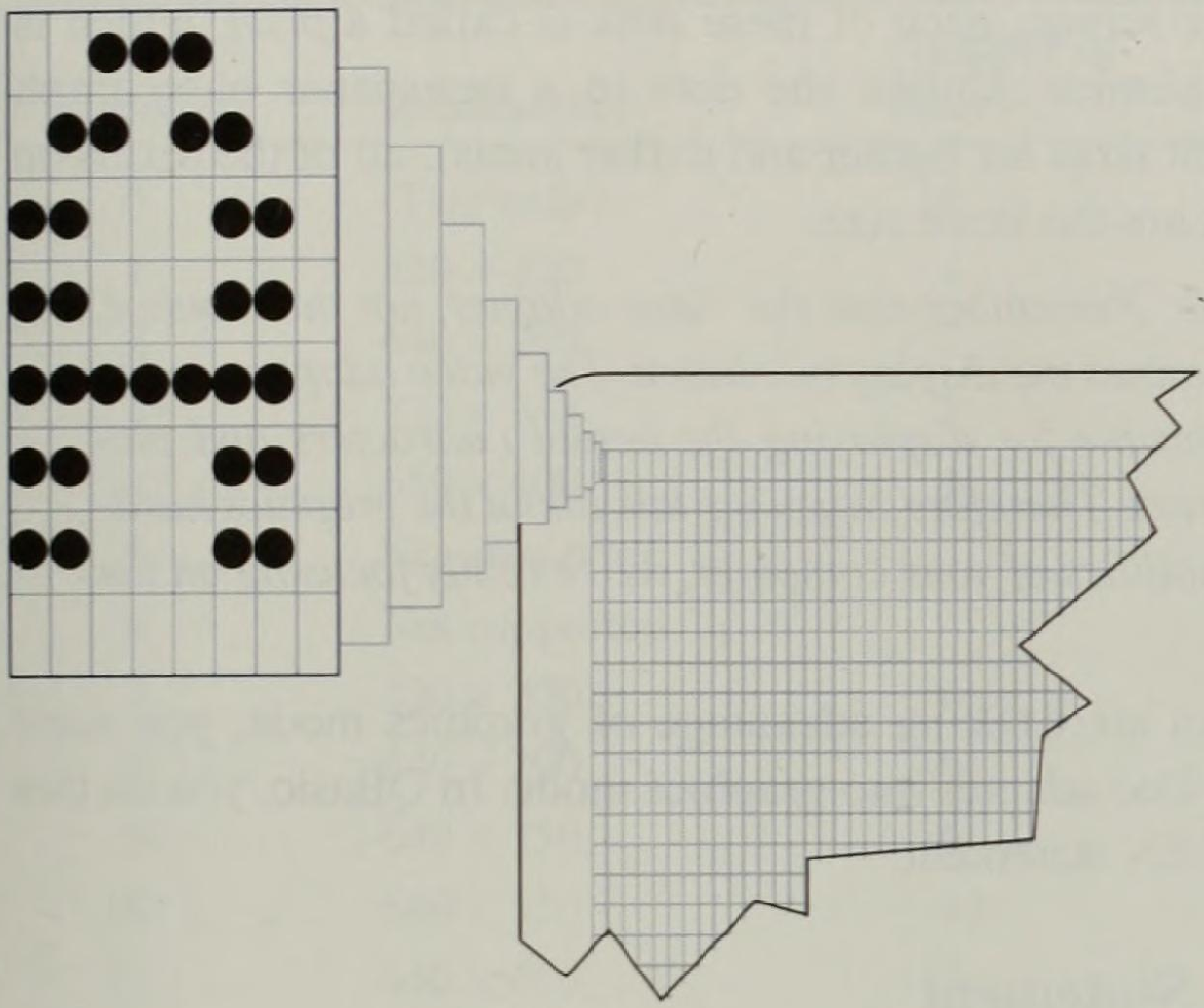
Output such as smoothly drawn shapes and flowing animation is better accomplished if your video adapter is in graphics mode and if you take advantage of the graphics tools that QBasic has to offer.

What Is Graphics Mode?

If you look closely at one of the characters on your screen, you'll see that it's actually made up of tiny dots that collectively form the letter. In fact, each of these ''boxes'' that we've been talking about is actually a matrix (a grid) of 8 dots by 8 dots, as shown on the next page.

In the sample matrix, the letter A is filled in to show how a matrix of dots displays a character. Although not all the dots are used, *all* the dots in the matrix—filled in or not—appear on the screen when your video adapter is in text mode. This is why an entire ''box'' is needed for each character.

In graphics mode the rules change. In graphics mode you can access each individual dot.



NOTE: Some video adapters can display matrices of different sizes. For example, some can display matrices 8 dots wide and 9 dots high; others can display matrices 9 dots wide and 14 dots high, and so forth. We'll assume an 8-by-8 matrix for this discussion.

Graphics resolution

In computer graphics, the word *resolution* is defined as the fineness of detail in an image. Generally speaking, a low-resolution image has a grainy appearance, and a high-resolution image has a smooth, almost photographlike appearance. The higher the resolution, the better the image quality. If you look closely at a newspaper photograph, you'll see that the picture is actually made up of thousands of tiny dots. Hold the picture far away, and collectively these seemingly random dots form a recognizable image.

A computer screen (or a regular television set, for that matter) also forms pictures by displaying tiny dots that collectively form an image. The smaller the dots—that is, the greater the resolution—the better the picture quality.

On a computer screen, each of these dots is called a *pixel,* which is short for *picture element.* Unlike the dots in a newspaper photograph (which are different sizes for lighter and darker areas), all of the pixels on a computer screen are the same size.

NOTE: Remember that the video adapter, not the monitor, determines the display resolution. The video adapter is also responsible for displaying the actual characters and pictures on the screen. Therefore, when we talk about the graphics hardware and capabilities of your computer, we're really focusing on the video adapter.

Before you can start taking advantage of graphics mode, you must first switch your video adapter into graphics mode. In QBasic, you do this by using the SCREEN statement.

The SCREEN Statement

The SCREEN statement causes QBasic to place your video adapter in graphics mode when it displays output. The simplest form of the SCREEN statement is as follows:

```
SCREEN mode
```

mode is a number from 0 through 13 (except 5 and 6) that tells QBasic what graphics mode you want it to switch your video adapter to. Although 12 graphics modes are available, your video adapter can use only some of them. And your video adapter can display in only one mode at a time.

Each mode is unique in terms of its resolution and the number of colors it supports: Some modes display 25 rows and 80 columns of text; others display 25 rows and 40 columns. And some modes display more than the usual 25 rows of text.

Table 11-2 summarizes the 14 screen modes. The resolutions shown represent the maximum possible resolutions for particular modes. The values are given as *horizontal* × *vertical,* so the values 320 × 200 mean that the display is 320 pixels wide and 200 pixels high.

The number of colors shown for each mode represents the maximum number of colors available in that mode. Note that some adapters might be unable to display the maximum number of colors for a particular mode, even if the adapter is capable of displaying that mode.

Screen mode	*Resolution*	*Number of colors*	*Text resolution*
0	(Text only)	16	80 × 25
1	320 × 200	4	40 × 25
2	640 × 200	2	80 × 25
3*	720 × 348	1	80 × 25
4†	640 × 400	1	80 × 25
5	Not supported		
6	Not supported		
7	320 × 200	16	40 × 25
8	640 × 200	16	80 × 25
9	640 × 350	16	80 × 25
10	640 × 350	4‡	80 × 25
11	640 × 480	2	80 × 30
12	640 × 480	16	80 × 30
13	320 × 200	256	40 × 25

* Hercules graphics cards and compatibles only.

† Olivetti personal computers and AT&T 6300 personal computers only.

‡ Mode 10 shows different *attributes* instead of different colors. In other words, images are shown in a regular, bright, or blinking form instead of in different colors.

TABLE 11-2.
The available screen modes and their characteristics.

The text resolution shown for each mode indicates the size in which text will be displayed if you use PRINT statements in your program. Note that most modes use 25 rows; more important at this point is the number of columns of text a particular mode will display.

Practice: Identifying the adapter in your computer

Type in the TEST-VID.BAS program (Figure 11-6 on the next page) and run it. This program provides you with a list of the mode settings you can use with the SCREEN statement. You might want to make a mark in Table 11-2 beside each mode supported by your particular video adapter.

```
TEST-VID.BAS
' This program determines what screen modes your video adapter supports.

DIM okModes%(18)               ' array to hold video modes being tested

CLS

ON ERROR GOTO CannotSetMode:   ' tell QBasic what to do in case
                               '   it encounters an error

FOR i% = 0 TO 13               ' try to set the video adapter in
    SCREEN i%                  '   all 14 modes; if an error occurs
    IF errorCode% = 0 THEN     '   (meaning that mode isn't supported),
        okModes%(i%) = 1       '   then jump to the end of the program
        bestMode% = i%
    END IF
    errorCode% = 0
NEXT i%

SCREEN bestMode%               ' set best video mode for display

PRINT bestMode%
FOR i% = 0 TO 13
    IF okModes%(i%) = 1 THEN
        PRINT "You can use mode"; i%; "with your computer."
    END IF
NEXT i%

END

CannotSetMode:           ' "dummy" routine to handle error trapping
    errorCode% = 1       '   (to prevent QBasic from stopping and
    RESUME NEXT          '   displaying an error message)
```

FIGURE 11-6.
TEST-VID.BAS: A program that tests your video adapter.

NOTE: The TEST-VID.BAS program uses two new statements—ON ERROR GOTO and RESUME NEXT—to tell QBasic what to do if it encounters a video mode your adapter doesn't support. Beyond this simple use, ON ERROR GOTO and RESUME NEXT are beyond the scope of this book. Use the online help to find out more about these statements.

Understanding Graphics-Mode Coordinates

With the LOCATE statement you learned that by specifying row and column coordinates you could dictate where text would appear on the output screen. In graphics mode, the output screen actually has two types of coordinates, depending on the type of information you want to display:

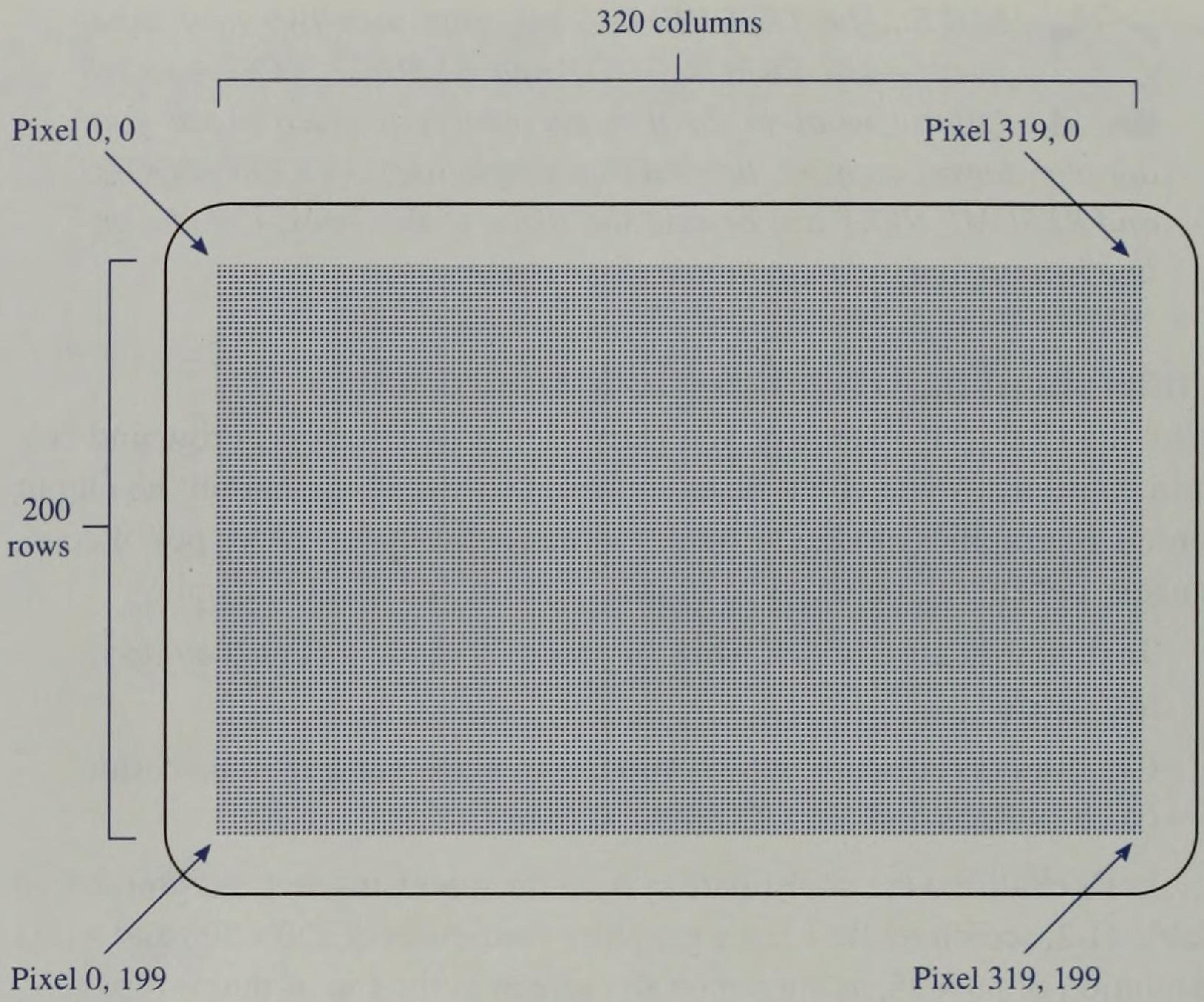
- Text coordinates, which work in graphics mode just as they do in text mode
- Graphics coordinates, which you use to place nontext information (such as lines and circles) on the screen

Let's examine the coordinate system for screen mode 1. As you saw in Table 11-2, screen mode 1 has a graphics resolution of 320 × 200 and a text resolution of 40 × 25, as shown on the screen at the top of the next page.

Notice one very important difference between the graphics coordinate system and the text coordinate system: The upper-left pixel on the graphics screen is at coordinates (*0*, *0*), and the upper-left character on the text screen is at coordinates (*1*, *1*).

Another important difference is in how you specify the coordinates: In text mode, you give the coordinates as *row*, *column*; in graphics mode, you specify the coordinates as *column*, *row*.

Notice also that in graphics mode, just as in text mode, each screen location has unique coordinates.



Working with Individual Pixels

After you've used the SCREEN statement to put your video adapter into graphics mode, you can use two QBasic statements to place individual pixels on the screen at specific locations: PSET and PRESET.

The PSET statement

In its simplest form, the syntax of a PSET statement is as follows:

```
PSET (xcoordinate, ycoordinate)[, color]
```

xcoordinate is the number of the column in which you want a pixel to appear. The range of values you can use depends on the graphics screen mode you've set with the SCREEN statement. For example, if you chose screen mode 1, the *xcoordinate* value would be from 0 through 319 because the maximum number of columns in screen mode 1 is 320.

ycoordinate is the number of the row in which you want the pixel to appear. Again, the range of values you can use depends on the graphics screen mode you've set using the SCREEN statement. For example, if you set screen mode to 1, the *ycoordinate* value would be from 0 through 199 because the maximum number of rows in screen mode 1 is 200.

NOTE: The parentheses around the xcoordinate *and* ycoordinate *values are not optional. You must use the parentheses around these values in the PSET statement.*

color is a value representing the color of the pixel. Again, the range of color values you can choose from depends on the graphics screen mode you've set with the SCREEN statement. If you omit *color*, QBasic displays the pixel in the current foreground color.

A PSET statement sets only one pixel at a time, but you can put a PSET statement inside a loop to draw several pixels in a row.

To erase a pixel drawn with the PSET command, put the coordinates of the pixel you want to erase in another PSET command and specify the background color for the *color* value.

Practice: Using the PSET statement

1. Type in the PSET.BAS program (Figure 11-7).

```
' PSET.BAS
' This program demonstrates the PSET statement.

CLS

INPUT "Please enter the screen mode (0-13):  ", modeNum%
SCREEN modeNum%

LOCATE 22, 23
PRINT "Col: Row:"
```

FIGURE 11-7. *(continued)*
PSET.BAS: A demonstration of the PSET statement.

FIGURE 11-7. *continued*

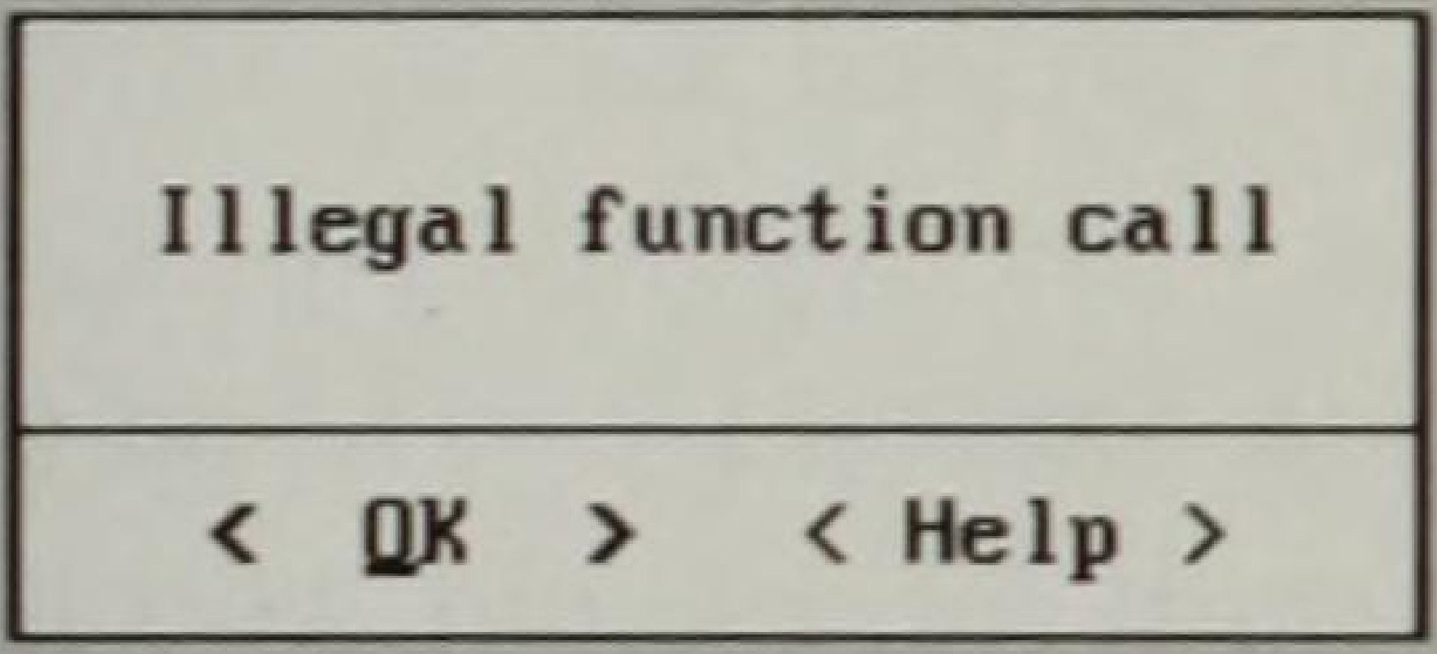
```
FOR i% = 0 TO 200 STEP 20          ' values for xcoordinate
    FOR j% = 0 TO 135 STEP 15      ' values for ycoordinate
        PSET (i%, j%)              ' use default foreground color
        LOCATE 23, 1               ' position text cursor
        PRINT "Current coordinates: ("; i%; ","; j%; ")"

        FOR k% = 1 TO 2000         ' delay loop
        NEXT k%

        PSET (i%, j%), 0           ' use background color to erase
        LOCATE 23, 22
        PRINT SPACE$(13)           ' erase printed coordinates
    NEXT j%
NEXT i%
```

2. Run the program and enter a screen value from the list of values you can use with your particular video adapter (the list you obtained in the last practice). The program will then begin to "move" a pixel in regular jumps across the screen. After the program places each pixel on the screen, the program updates the message at the bottom of the screen to show you the pixel's coordinates.

   If you can use more than one screen mode with your adapter, run the program again and enter a different mode number. Notice the difference in resolution—the program will almost cover a low-resolution screen (such as 320 × 200) with pixels but will cover only the upper-left corner of a high-resolution screen (such as 640 × 480).

Illegal Function Calls

Inevitably, you'll try to run a program that includes graphics or sound statements only to have QBasic return you to the View window and display the following dialog box:



Illegal function call

In addition, QBasic will highlight the statement it had trouble executing. For example, if you had tried to run the preceding program with a screen mode of 0, QBasic would have stopped the program, returned you to the View window, displayed the dialog box, and then highlighted the first PSET statement. Why?

If you look at Table 11-2, you'll notice that screen mode 0 is text-only, meaning that you can't use graphics statements (such as PSET) with your video adapter in that mode.

Other conditions can cause a similar error:

- Trying to use a COLOR statement value that is not supported by the screen mode
- Trying to use a LOCATE statement value of 0 or a value greater than the available number of rows or columns
- Trying to use a SCREEN statement value that your video adapter does not support

If you see this dialog box, review Table 11-2, examine the list of valid video modes that you obtained from the TEST-VID.BAS program, and then correct your program.

Changing the foreground and background colors

Chapter 6 provided a brief introduction to the COLOR statement. A quick review is in order at this point.

The COLOR statement has the following syntax:

```
COLOR [foreground][, background]
```

foreground and *background* are values for the foreground and background colors, respectively. The range of values you can use depends entirely on the graphics mode your video adapter is in. (See Table 11-2.) If you omit either value, the current value is used. If you omit *foreground*, you must precede *background* with a comma.

The foreground value determines the color of text and graphics shapes. You can change the foreground color at any point in the program without affecting text or shapes already displayed on the screen.

The background value specifies the background color. (You can change the background color in modes 0, 7, 8, 9, and 10 only.) Note that changing the background color will not have an immediate effect. To change the color of the entire background to the new color, you must include an instruction in your program that updates the background color. The CLS statement is an excellent candidate, although you should do this early in your program because a CLS statement will erase any text or graphics already displayed on the screen.

The PRESET statement

The PRESET statement works the way the PSET statement does, with one important reversal: If you omit the *color* argument, QBasic displays the pixel in the current background color. This lets you erase pixels simply by omitting *color*. The PRESET statement has the following syntax:

```
PRESET (xcoordinate, ycoordinate)[, color]
```

Note that the PRESET statement erases whatever happens to be located at (*xcoordinate, ycoordinate*). For example, if you use a PRINT statement to print a message and then use a PRESET statement whose coordinates are in the same area as the message, the PRESET statement might ''take a bite'' out of a letter in the message.

Practice: Working with the PRESET statement

1. Type in the PRESET.BAS program (Figure 11-8).

```
' PRESET.BAS
' This program demonstrates the PRESET statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
INPUT "Press Enter to begin the hailstorm...", dummy$

SCREEN modeNum%
PRINT "Press any key to stop"

DO
    RANDOMIZE TIMER
    randCol% = INT(RND(1) * 320)  ' random column number for "hailstones";
                                  '   assumes horizontal resolution of 320;
                                  '   change for higher-resolution modes

    FOR i% = 1 TO 199             ' assumes vertical resolution of 200;
                                  '   change for higher-resolution modes
        PSET (randCol%, i%)
        PRESET (randCol%, i% - 1)

        FOR j% = 1 TO 35          ' delay loop
        NEXT j%

    NEXT i%
LOOP UNTIL INKEY$ <> ""
```

FIGURE 11-8.
PRESET.BAS: A demonstration of the PRESET statement.

2. Run the program. If your video adapter allows it, enter the value *1*, *7*, or *13* for maximum effect. The comments in the program show you what to change if you want the program to take advantage of higher-resolution screen modes.

As the program runs, you can observe one of the interesting side effects of PRESET. Just before the ''hailstones'' appear, the program prints the message *Press any key to stop.* As the program continues to run, you'll see the PRESET statements demolish this message as the ''hailstones'' pass over it.

Note the use of the INKEY$ function, introduced here for the first time. INKEY$ checks the keyboard to see whether a key has been pressed. If so, INKEY$ returns the ASCII code associated with the key; if not, it returns an empty string. Unlike the INPUT and LINE INPUT statements, the INKEY$ function doesn't wait for the user to press Enter before it continues.

Positioning Pixels with Coordinates

You can use two coordinate systems to position pixels on the screen: *absolute coordinates* and *relative coordinates.*

Absolute coordinates

The coordinate system you've used so far is the absolute coordinate system. In this system, all coordinate values are based on the upper-left-corner coordinate, (*0*, *0*). To position a pixel on the screen, you count (or, more likely, guess) the number of columns over and the number of rows down in order to come up with the coordinate values of the pixel position.

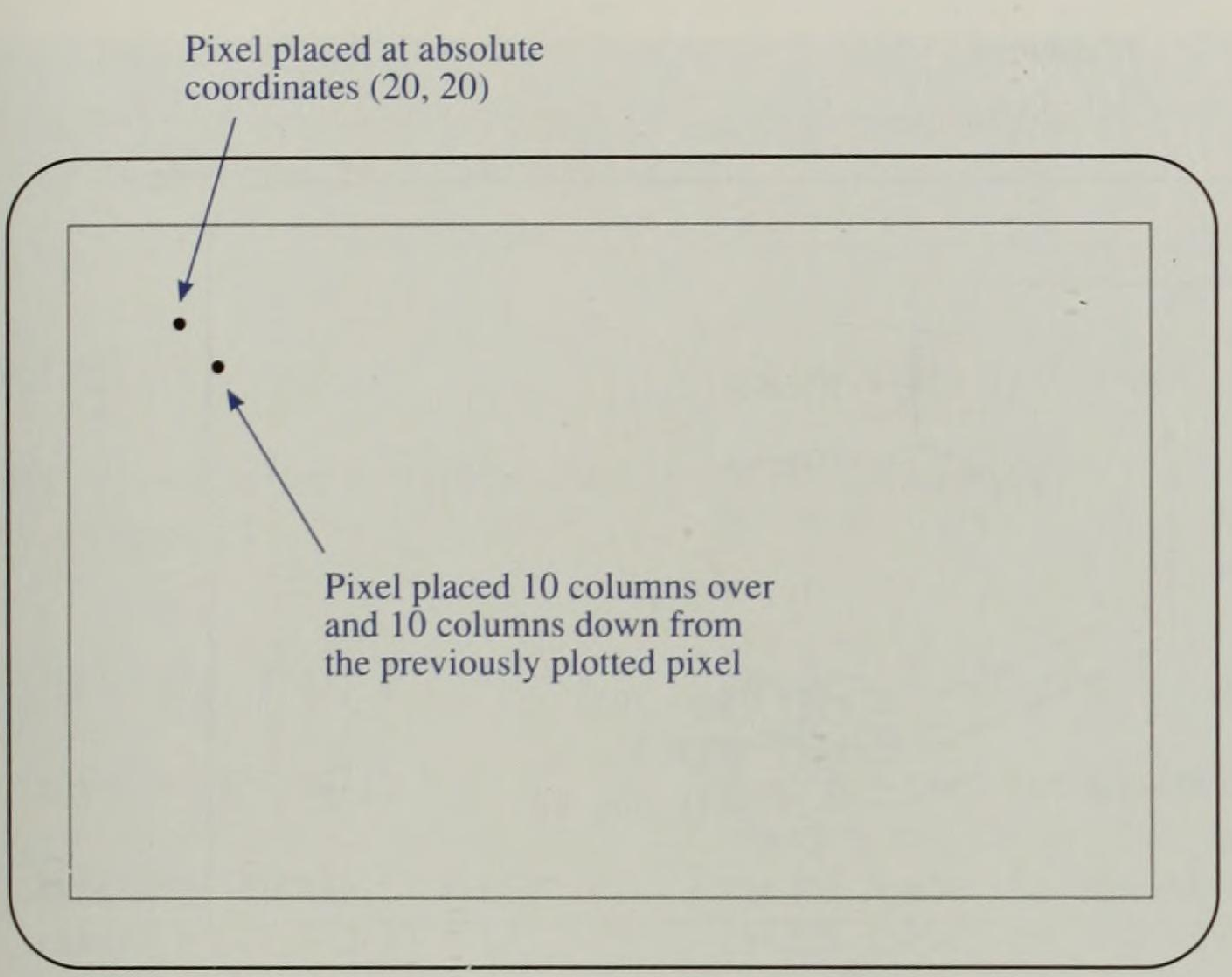
As you've seen, the absolute coordinate system works fine, but if you want to use individual pixels to create a particular shape, you're in for a lot of guesswork and counting. The relative coordinate system makes the job of determining coordinate values much easier.

Relative coordinates

With relative coordinates, the coordinates you specify for a specific pixel are *relative* to the last pixel you placed on the screen.

The illustration at the top of the next page demonstrates this. To place the first pixel on the screen, you must use the absolute coordinates because no relative pixel location exists. For the second and subsequent pixels, however, you can use relative coordinates.

To use the relative coordinate system, you use the STEP keyword within a PSET or PRESET statement.


Pixel placed at absolute coordinates (20, 20)
Pixel placed 10 columns over and 10 columns down from the previously plotted pixel

The STEP keyword

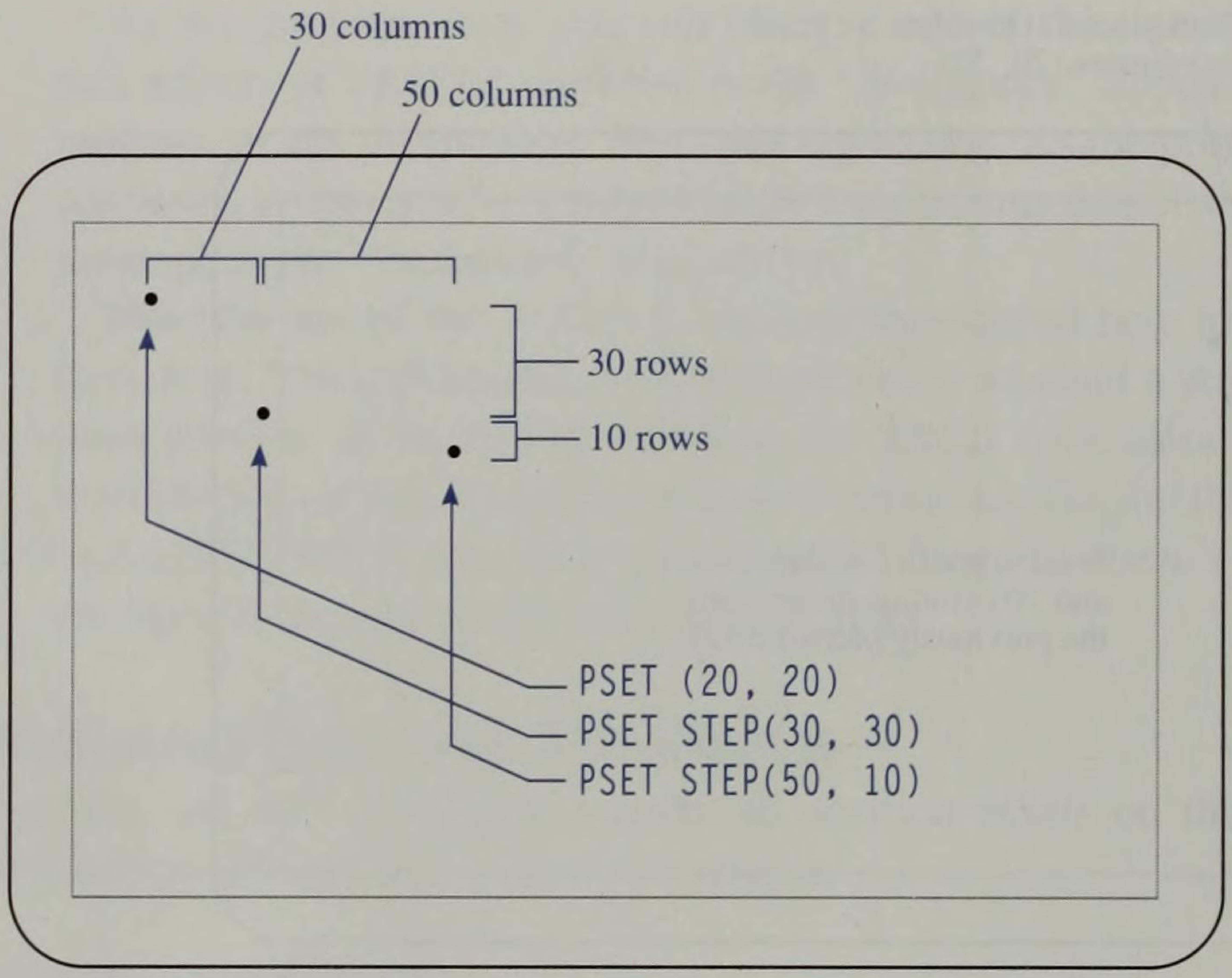
The syntax of a PSET statement that contains the STEP keyword is as follows:

```
PSET STEP(xcoordinate, ycoordinate)[, color]
```

(The syntax is identical for a PRESET statement that contains the STEP keyword; simply substitute the PRESET keyword for the PSET keyword.)

The *xcoordinate* and *ycoordinate* values are positive or negative numbers that tell QBasic where to position the pixel in relation to the last pixel it positioned with a PSET or PRESET statement. If no previous pixel exists, QBasic uses the absolute coordinates (*0*, *0*).

The screen illustration at the top of the next page demonstrates this ‘‘daisy-chain’’ effect: The location of each pixel but the first is dependent on the location of the previous pixel. Always keep in mind that the STEP keyword causes QBasic to position a pixel relative to the location of the *most recent* pixel it positioned.



Practice: Working with the STEP keyword

1. Type in the STEP-1.BAS program (Figure 11-9).
2. Run the program and enter a screen value. The program assumes a resolution of 320 × 200, so use a value of *1*, *7*, or *13* if possible. (Check what's supported by your video adapter.) Press any key to stop the program.

This program provides the illusion of traveling through space by simulating ''stars'' zooming by. Inside the DO loop, the program sets an ''invisible'' pixel at the center of the screen. This is the reference point on which later PSET statements base their coordinates.

Next, the random number *quad%* determines which quadrant the star will zoom through. Note the two IF statements—one for quadrants 0 and 1 and another for quadrants 0 and 3. (Quadrant 2 uses positive values, so no modification is necessary.) The values modified in the IF statements affect the PSET coordinates inside the FOR loop—some use positive coordinates, some use negative coordinates, and some use both.

```
' STEP-1.BAS
' This program uses the illusion of zooming through space
'   to demonstrate the STEP keyword.

CLS

CONST DELAY% = 200                  ' controls how fast stars "shoot"

INPUT "Please enter a screen mode (0-13):  ", modeNum%
SCREEN modeNum%

DO
    PSET (160, 100), 0              ' set center (assumes 320 by 200
                                    '   resolution)
    quad% = INT(RND(1) * 4)         ' random number for quadrant from
                                    '   which the "star" will shoot
    randX% = INT(RND(1) * 10)       ' random number for column movement
    randY% = INT(RND(1) * 10)       ' random number for row movement

    IF quad% = 0 OR quad% = 1 THEN  ' normal movement is down;
        randY% = -randY%            '   reverse y value to move up
    END IF
    IF quad% = 0 OR quad% = 3 THEN  ' normal movement is to right;
        randX% = -randX%            '   reverse x value to move to left
    END IF

    FOR i% = 1 TO 20
        PSET STEP(-randX% * i%, randY% * i%) ' draw "star"

        FOR j% = 1 TO DELAY%        ' delay loop
        NEXT j%

        PRESET STEP(0, 0)           ' erase "star"
    NEXT i%

LOOP UNTIL INKEY$ <> ""
```

FIGURE 11-9.
STEP-1.BAS: Zooming through space using the STEP keyword.

In the PSET statement, a random number is multiplied by the current value of the loop variable *i%*. With each loop the resulting value increases, causing the ''star'' to move farther from the center of the screen. The ''star'' starts off slowly in the center and accelerates as it gets closer to the edge of the screen.

Finally, note how the PRESET statement is used. The STEP keyword causes the PRESET statement to use relative coordinates—coordinates determined by the PSET statement. Because the coordinates provided are (*0*, *0*), the PRESET statement causes QBasic to place the PRESET pixel at the same location as the preceding PSET pixel—effectively erasing it.

Just for fun, think about how you might write this program with absolute coordinates. You can do it (you might even enjoy the challenge), but as you'll soon discover, relative coordinates are better suited to this particular programming feat.

CREATING COMPLEX SHAPES

Plotting individual pixels gives you precise control as you create graphics on the output screen. However, for complex shapes such as lines, boxes, circles, and polygons, you would have a lot of coordinate calculating to do. For a line, you could use a loop to quickly plot a series of individual points, but you'd still be faced with a lot of calculation. Fortunately, QBasic provides a set of tools that make the job of creating complex shapes a snap.

The LINE Statement

The LINE statement offers a single-statement way to create lines. If you use the LINE statement, QBasic does most of the work for you.

Drawing a simple line

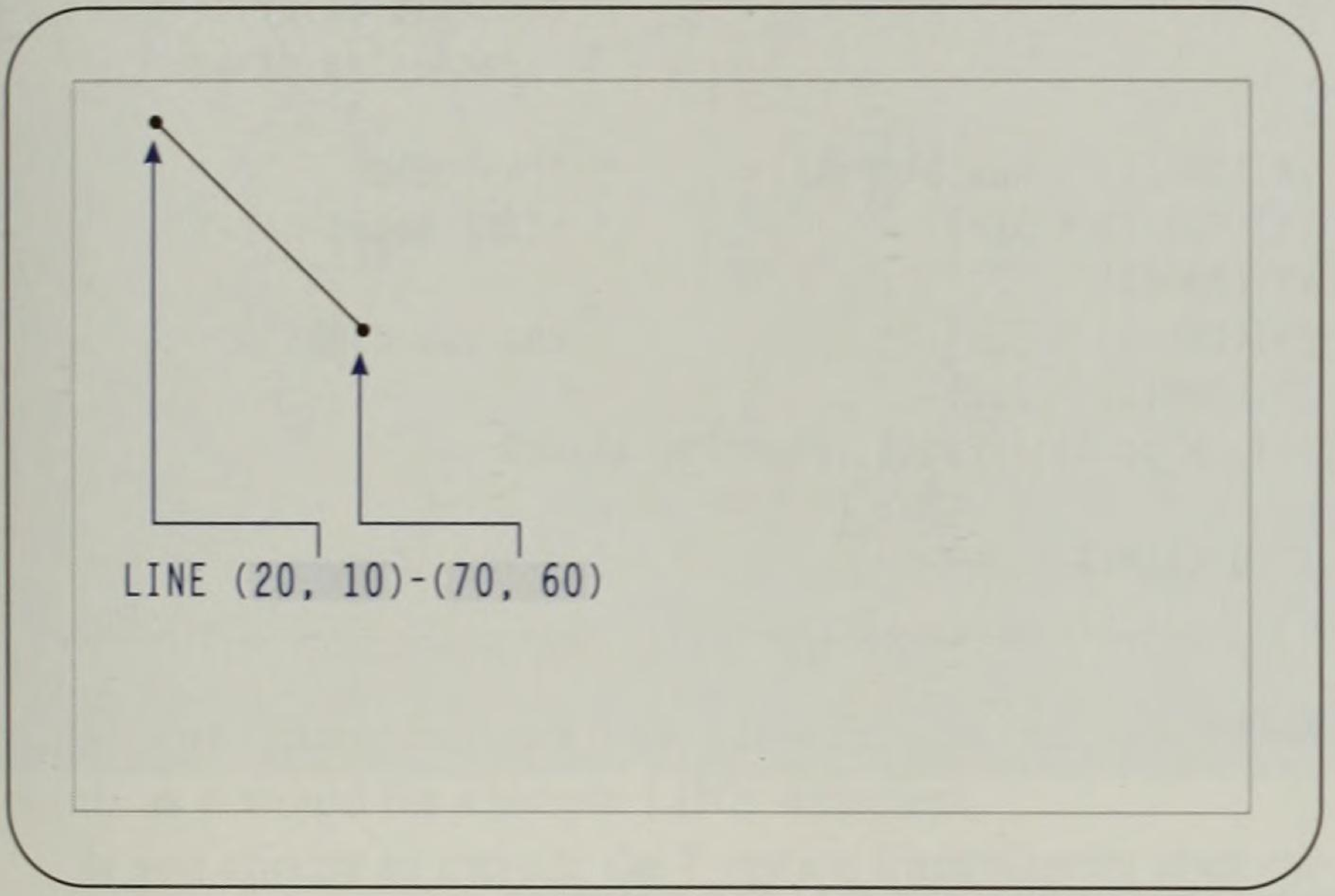
In its simplest form, the LINE statement uses the following syntax:

```
LINE (x1, y1)-(x2, y2)[, color]
```

Note that unlike the PSET and PRESET statements, which use only a single coordinate set, a LINE statement uses *two* sets of coordinates:

- *x1* and *y1* are the column and row coordinates of the starting point of the line.
- *x2* and *y2* are the column and row coordinates of the ending point of the line.

Here's an example of how the LINE statement works:



The starting point needn't be to the left of the ending point. In fact, you can start the line in the lower-right corner of the screen and work upward and to the left if you want. Simply specify the coordinates, and QBasic "connects the dots."

As with any other graphics command, the range of valid values for *x1*, *y1*, *x2*, and *y2* is based on the mode of your video adapter. You must enclose the coordinates in parentheses and separate the sets with a hyphen.

Practice: Working with the LINE statement

1. Type in the LINE-1.BAS program (Figure 11-10 on the next page).

```
' LINE-1.BAS
' This program demonstrates the LINE statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
INPUT "How many colors?  ", numColors%
SCREEN modeNum%

CONST DELAY% = 100                            ' controls delay after
                                              '   each line drawn
DO
    shade% = INT(RND(1) * numColors%) + 1     ' line color
    x1pos% = INT(RND(1) * 320)                ' start coordinates
    y1pos% = INT(RND(1) * 200)
    x2pos% = INT(RND(1) * 320)                ' end coordinates
    y2pos% = INT(RND(1) * 200)
    LINE (x1pos%, y1pos%)-(x2pos%, y2pos%), shade%

    FOR i% = 1 TO DELAY%                      ' delay loop
    NEXT i%

LOOP WHILE INKEY$ = ""
```

FIGURE 11-10.
LINE-1.BAS: Drawing random lines with the LINE statement.

2. Run the program. The program asks you for a screen mode and a maximum number of colors and then prints random lines on the screen in various colors. Press any key to stop the program.

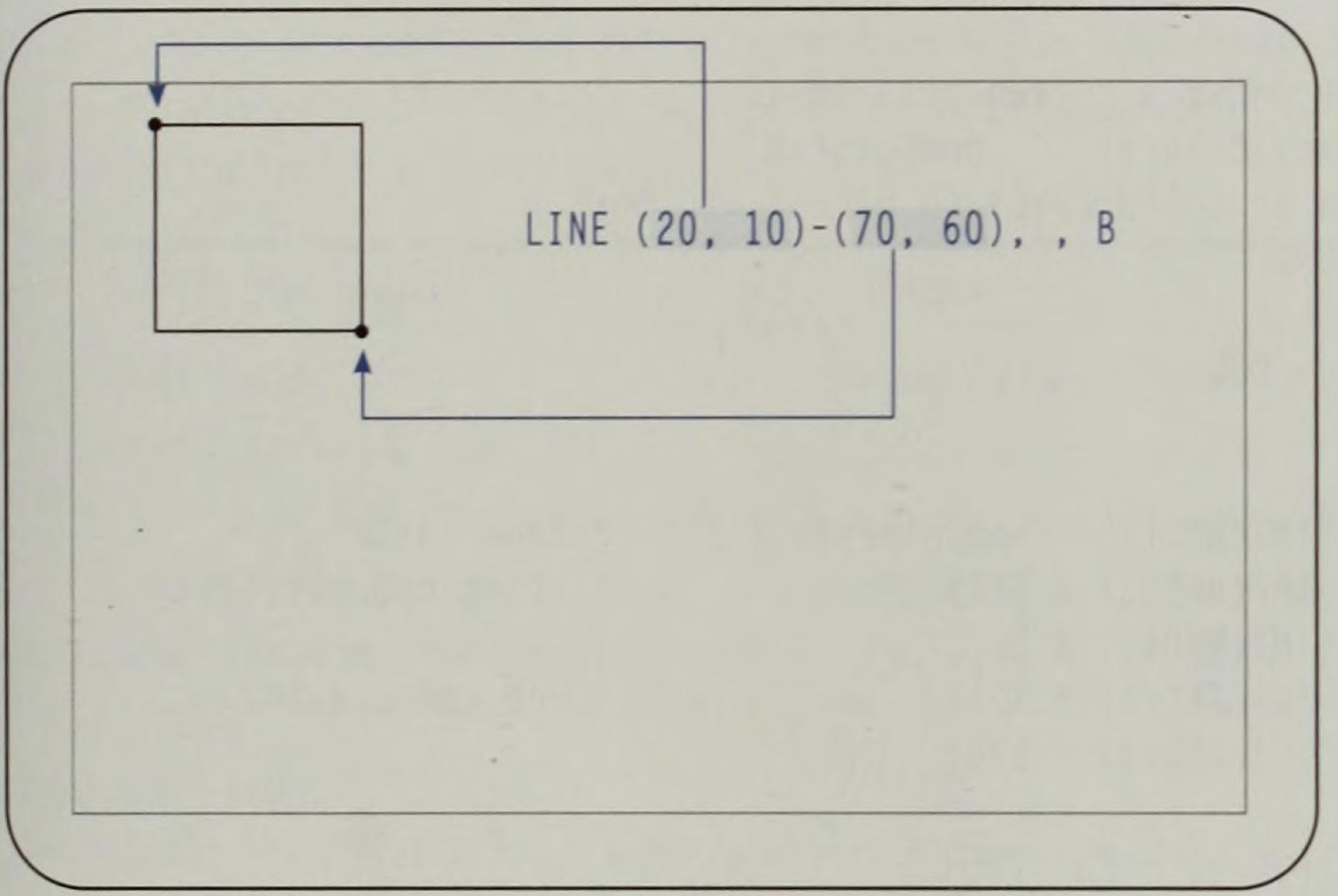
Drawing a box

Drawing a box, hollow or filled, is as easy as drawing a single line. You could use four LINE statements to accomplish the job, but the inventors of Basic figured that boxes were common enough to warrant their own feature in the Basic language—saving you considerable time and calculation.

Interestingly enough, you use a LINE statement to create a box:

```
LINE (x1, y1)-(x2, y2)[, [color][, [B[F]]]]
```

This is the same LINE statement you just learned about, with one important difference. If you add the B option at the end of the statement, QBasic draws a box using the start and end positions as opposite corners of the box. Here's how this works:



Notice that QBasic does *not* draw a line directly between the start and end points as it would for a simple LINE statement.

If you choose to include the F option immediately after the B option, QBasic fills the box with the current foreground color. You can use the F option only if you use the B option.

Practice: Working with hollow and filled boxes

1. Type in the LINE-2.BAS program (Figure 11-11 on the next page).
2. Run the program. Based on your responses, the program creates either a random pattern of hollow boxes or a random pattern of filled boxes. As this program shows, the boxes can be any size or shape and you can put them anywhere on the screen.

```
' LINE-2.BAS
' This program demonstrates the box-drawing capabilities
'   of the LINE statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
INPUT "How many colors?  ", numColors%
INPUT "Hollow or solid boxes (H or F)?  ", box$
SCREEN modeNum%

CONST DELAY% = 800

DO
    shade% = INT(RND(1) * numColors%) + 1    ' line color
    x1pos% = INT(RND(1) * 320)               ' start coordinates
    y1pos% = INT(RND(1) * 200)
    x2pos% = INT(RND(1) * 320)               ' end coordinates
    y2pos% = INT(RND(1) * 200)

    IF UCASE$(box$) = "H" THEN
        LINE (x1pos%, y1pos%)-(x2pos%, y2pos%), shade%, B
    ELSEIF UCASE$(box$) = "F" THEN
        LINE (x1pos%, y1pos%)-(x2pos%, y2pos%), shade%, BF
    END IF

    FOR i% = 1 TO DELAY%                     ' delay loop
    NEXT i%

LOOP UNTIL INKEY$ <> ""
```

FIGURE 11-11.
LINE-2.BAS: Drawing random boxes with the LINE statement.

Drawing complex shapes

The lines, hollow boxes, and filled boxes you just worked with all used absolute coordinates. As with PSET and PRESET, you can use the STEP keyword with the LINE statement to specify relative coordinates for a line or a box. Or you can omit the starting point of the line or box—forcing QBasic to use the ending point of the last-drawn object (pixel or line) as the starting point for the new line.

Here is the syntax of a LINE statement that uses relative coordinates:

```
LINE [[STEP](x1, y1)]-[STEP](x2, y2)[, [color][, [B[F]]]]
```

Using relative coordinates allows you to create complex line drawings that would take some time to calculate if you used absolute coordinates. Figure 11-12 shows some sample LINE statements that use both absolute and relative coordinates. You can see the result of each statement. In each case, assume that the last point was set at *(20, 20)*.

LINE statement	*Result*
`LINE -(30, 50)`	Draws from (20, 20) to (30, 50)
`LINE -STEP(30, 50)`	Draws from (20, 20) to (50, 70)
`LINE (40, 40)-STEP(60, 60)`	Draws from (40, 40) to (100, 100)
`LINE STEP(30, 50)-(70, 70)`	Draws from (50, 70) to (70, 70)
`LINE STEP(30, 50)-STEP(70, 70)`	Draws from (50, 70) to (120, 140)

FIGURE 11-12.
The results of LINE statements when used with absolute and relative coordinates.

Here is a summary of some conclusions you can draw from Figure 11-12:

- If you omit the starting coordinates, QBasic uses the coordinates of the most recent point as the starting coordinates.
- If you omit the starting coordinates and use the STEP keyword with the ending coordinates, QBasic positions the ending point *relative* to the most recent point's coordinates.
- If you include the starting coordinates and use the STEP keyword with the ending coordinates, QBasic positions the ending point *relative* to the specified starting coordinates.
- If you use the STEP keyword with the starting coordinates and do not use the STEP keyword with the ending coordinates, QBasic positions the starting coordinates *relative* to the coordinates of the most recent point and positions the ending point at the specified *absolute* coordinates.

- If you use the STEP keyword with both the starting and the ending coordinates, QBasic positions the starting coordinates *relative* to the coordinates of the most recent point and positions the ending point *relative* to the coordinates of the starting point it just calculated.

Using relative coordinates allows you to "daisy chain" lines and boxes together, simplifying the job of drawing complex shapes.

Practice: Working with LINE and relative coordinates

1. Type in the STEP-2.BAS program (Figure 11-13).

```
' STEP-2.BAS
' This program demonstrates the use of the STEP keyword in
'   the LINE statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
SCREEN modeNum%

blank$ = SPACE$(39)

PRINT "Enter 0 and 0 to quit"
PSET (150, 50)                 ' establish a starting point

DO
    LOCATE 22, 1
    INPUT "Horizontal movement (+ or -): ", horiz%
    INPUT "  Vertical movement (+ or -): ", vert%
    LINE -STEP(horiz%, vert%)
    LOCATE 22, 1
    PRINT blank$
    PRINT blank$
LOOP UNTIL horiz% = 0 AND vert% = 0
```

FIGURE 11-13.
STEP-2.BAS: A simple line-drawing program.

2. Run the program. This program is actually a simple drawing program. The program begins by plotting a pixel at (*150*, *50*), the starting point for your drawing. The bulk of the work is done in the DO loop. The program asks you to enter horizontal and vertical values, which can be either positive or negative. After you've entered these values, the program uses the LINE statement along with the STEP keyword to draw the line.

The CIRCLE Statement

QBasic also lets you create circles. By using the CIRCLE statement, you can create circles of various sizes that, in conjunction with the other graphics tools at your disposal, allow you to create interesting and useful graphics with your program.

In its simplest form, the syntax of a CIRCLE statement is as follows:

```
CIRCLE (xcoordinate, ycoordinate), radius[, color]
```

xcoordinate and *ycoordinate* are the same horizontal and vertical screen coordinates you've been working with. These values tell QBasic where to position the center of the circle. *radius* is a value that specifies, in pixels, the radius of the circle. (The radius of a circle is one-half its diameter.) *color* is an integer that tells QBasic what color to make the circle. If you omit *color*, QBasic uses the current foreground color.

As with any other graphics statement, the coordinates you use depend entirely on the graphics mode you set using the SCREEN statement.

Practice: Working with the CIRCLE statement

1. Type in the CIRCLE-1.BAS program (Figure 11-14 on the next page).
2. Run the program and enter an appropriate screen mode. The program then asks you to enter a column value, a row value, and a radius value. The CIRCLE statement in the middle of the program draws a circle based on the numbers you entered. Then, when you press Enter to continue, the CLS statement ''wipes the slate clean,'' and the process repeats until you enter *q* or *Q* to quit.

Try entering values greater than the maximum screen coordinates, or enter a relatively large radius value. You can use this technique to create a particular effect, such as an arc near one edge of the screen.

```
' CIRCLE-1.BAS
' This program demonstrates the CIRCLE statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
SCREEN modeNum%

DO
    LOCATE 21, 1
    INPUT "Please enter the column number:  ", colNum%
    INPUT "   Please enter the row number:  ", rowNum%
    INPUT "       Please enter the radius:  ", radius%
    CIRCLE (colNum%, rowNum%), radius%
    INPUT "Press Enter to continue or Q to quit: ", dummy$
    CLS
LOOP UNTIL UCASE$(dummy$) = "Q"
```

FIGURE 11-14.
CIRCLE-1.BAS: A simple circle-drawing program.

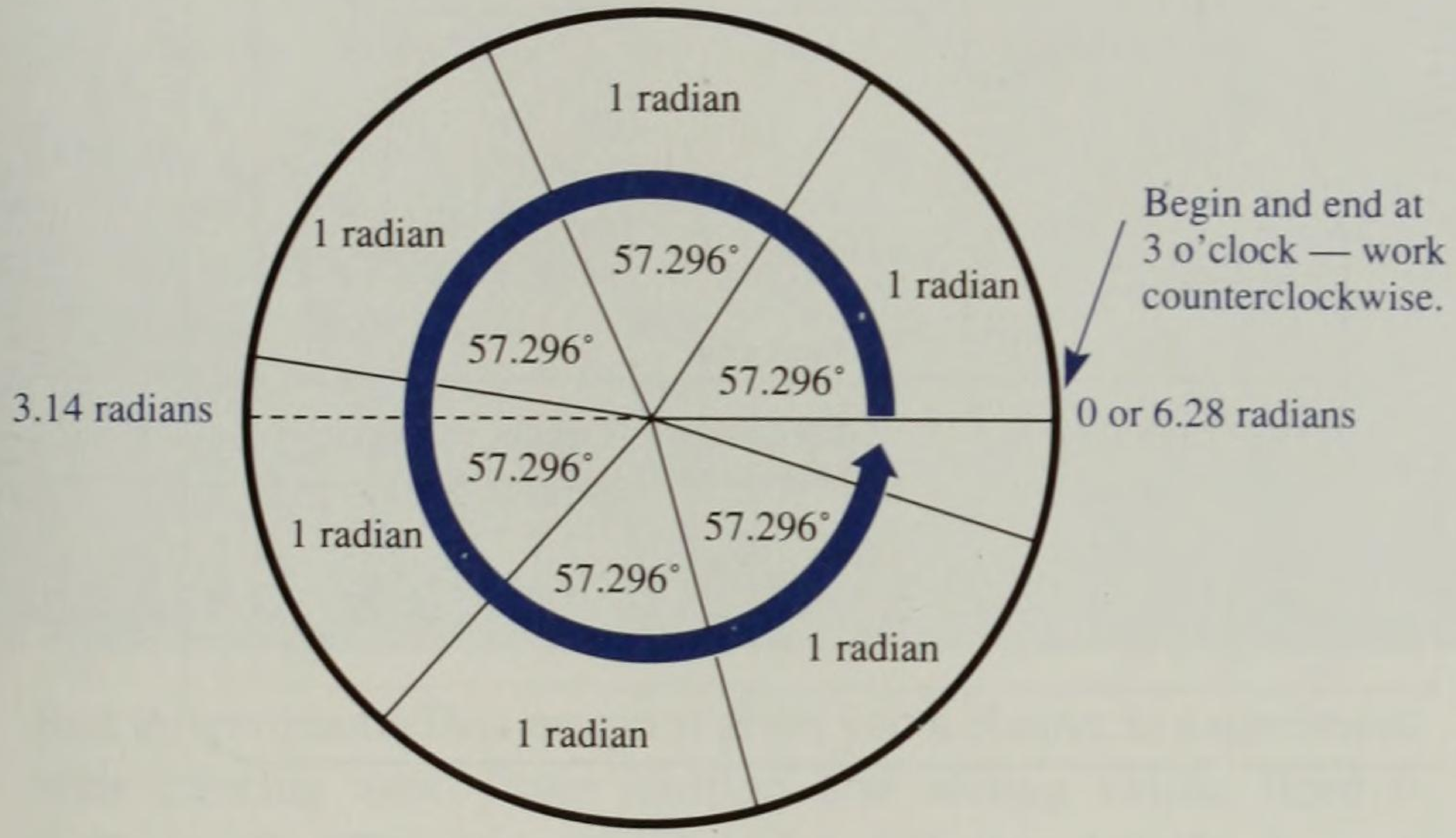
Drawing arcs

The CIRCLE statement also lets you draw arcs—segments of circles. Use a CIRCLE statement with starting and ending values to draw an arc:

CIRCLE (*xcoordinate*, *ycoordinate*), *radius*[, [*color*][, [*start*][, *end*]]]

This is the same syntax as that for the simple CIRCLE statement except that you add *start* and *end* values. The *start* and *end* values are the measurements, in radians, of the starting point and the ending point of the arc.

Although you might be more accustomed to measuring arcs in terms of *degrees,* the CIRCLE statement requires that you provide starting and ending measurements in *radians.* Figure 11-15 illustrates the relationship between the degree and the radian measurement systems. When drawing an arc of your own, simply refer to Figure 11-15 to determine the start and end points of your arc. We've included conversion information that you'll find helpful if you know the measurements of your arc in degrees.


1 radian
1 radian
57.296°
1 radian
Begin and end at 3 o'clock — work counterclockwise.
57.296°
57.296°
3.14 radians
0 or 6.28 radians
57.296°
1 radian
57.296°
57.296°
1 radian
1 radian

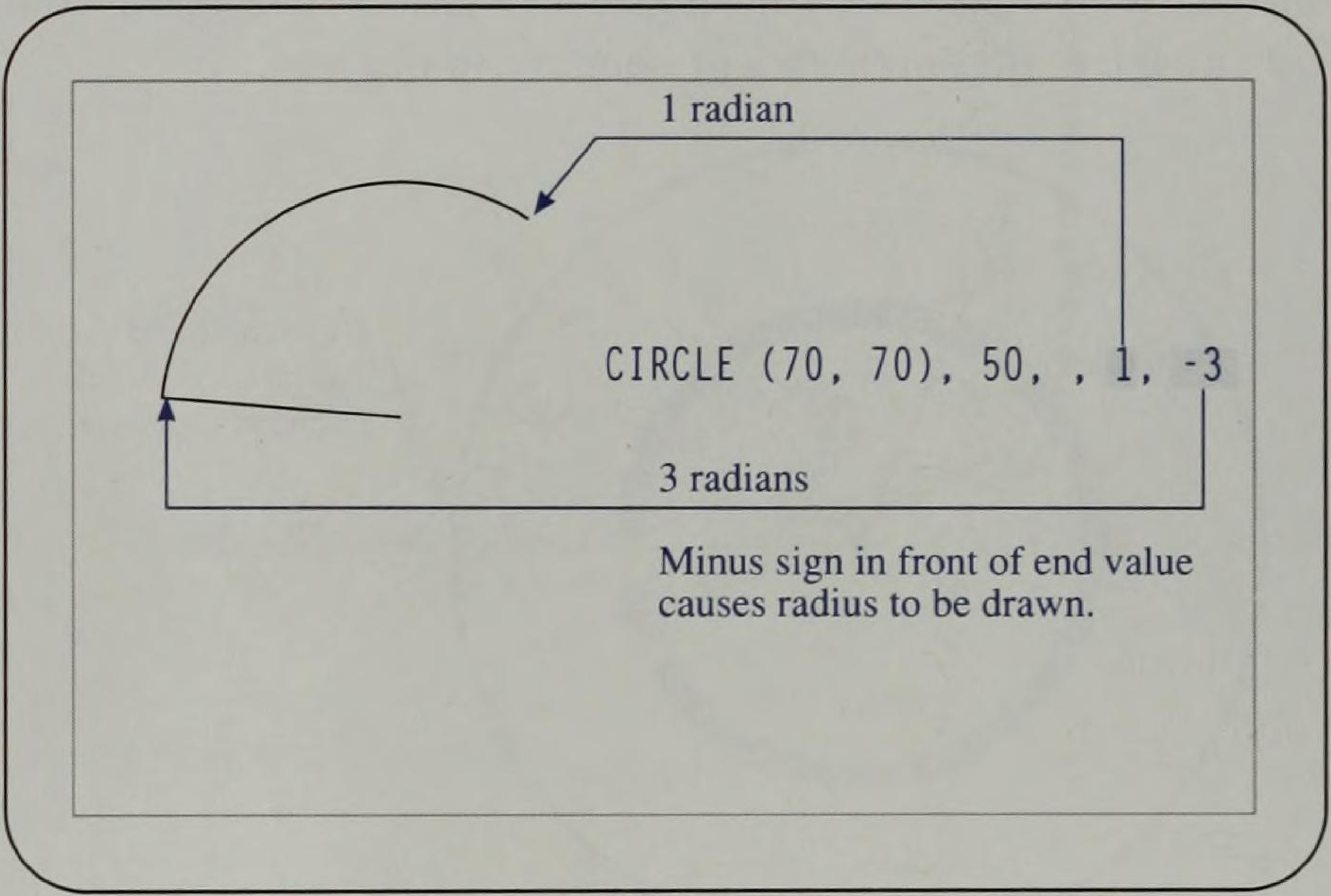
Radians = (3.14159 x (angle in degrees)) ÷ 180
1 degree = 0.0174532 radian

FIGURE 11-15.
Measuring an arc in radians.

Measurements start at 0 (3 o'clock) and progress counterclockwise to a maximum of 6.28 radians. The starting point 0 and the ending point 6.28 are the same location.

The values you provide for *start* and *end* are numbers from 0 through 6.28. (Note: The value 6.28 is technically 2 × π; we're using rounded numbers for simplicity.)

If you put a minus sign in front of the starting or the ending value, QBasic draws a line (the radius) from that point on the arc to the center of the "circle." The sample arc below shows a line from the ending point to the center of the "circle" because of the minus sign in front of the *end* value in the CIRCLE statement.



```
CIRCLE (70, 70), 50, , 1, -3
```

Minus sign in front of end value causes radius to be drawn.

Practice: Working with arcs

1. Type in the CIRCLE-2.BAS program (Figure 11-16).

```
' CIRCLE-2.BAS
' This program demonstrates how to create arcs with the CIRCLE statement.

CLS

INPUT "Please enter a screen mode (0-13):  ", modeNum%
SCREEN modeNum%
```

FIGURE 11-16.

CIRCLE-2.BAS: A program that creates arcs and pie wedges.

(continued)

FIGURE 11-16. *continued*

```
DO
    CLS
    PSET (150, 30): PSET (150, 125)  ' draw top and bottom points
    PSET (95, 78): PSET (205, 78)    ' draw left and right points
    LOCATE 2, 18: PRINT "1.57"       ' label for top point
    LOCATE 10, 7: PRINT "3.14"       ' label for left point
    LOCATE 18, 18: PRINT "4.71"      ' label for bottom point
    LOCATE 10, 28: PRINT "0 or 6.28" ' label for right point

    LOCATE 20, 1
    PRINT "Enter values between 0 and 6.28"
    PRINT "(A negative value draws a radius.)"
    INPUT "Please enter starting point:  ", starting!
    INPUT "Please enter ending point:  ", ending!
    CIRCLE (150, 78), 50, , starting!, ending!     ' no color value
    INPUT "Press Enter to continue or Q to quit: ", dummy$

LOOP UNTIL UCASE$(dummy$) = "Q"
```

2. Run the program. This program gives you a chance to experiment with creating arcs. Enter starting and ending values from 0 through 6.28. (Put a minus sign in front of a number if you want QBasic to draw a line from that point to the center of the circle.) Note that the starting value can be smaller than the ending value. Try several combinations of values to see how they affect the arcs the CIRCLE statement creates.

A LAST LOOK AT QBASIC GRAPHICS

We've covered quite a bit of ground here. QBasic offers some powerful and useful drawing tools. Regrettably, we can't cover them all here. But the tools you've become acquainted with in this chapter should keep you busy for quite some time and enable you to add some spiffy graphics to your own programs. Now you can add the enhancements we list on the next page to your programs.

- Bar charts that graphically illustrate data
- Pie charts that show the distribution of data in different categories
- Illustrations that make your programs more interesting
- Screen savers that protect your monitor from burn-in
- Animation that makes games and simulations more realistic

NOTE: You can always check the on-line help system for more information on graphics. Experiment—it's the best way to learn!

INTRODUCTION TO PROGRAMMING WITH SOUND

You've just learned how to use graphics to enliven your programs. Now you can learn how to jazz 'em up even more by using the sound capabilities of your computer and QBasic.

The SOUND Statement Revisited

In Chapter 6, where you learned about loops, you had a brief introduction to the SOUND statement. Now is an appropriate time for a quick review.

The SOUND statement has the following syntax:

```
SOUND frequency, duration
```

frequency is an integer value from 37 through 32,767 that specifies the frequency, in hertz (cycles per second), of the tone you want. Low frequencies create low tones; high frequencies create high tones. Note that the range of human hearing is approximately 20 to 20,000 hertz and that tones with frequencies above 16,000 hertz are more audible to dogs than they are to most people. Figure 11-17 lists the standard *frequency* values and their associated notes as established by the American Standards Association in 1936.

duration is an integer value that tells QBasic how long to play the tone. The *duration* value specifies for how many "clock ticks" the note should play—there are 18.2 clock ticks per second, so the *duration* value 9.1 would cause QBasic to play the note for one-half second.

You must specify both a *frequency* and a *duration* value in a SOUND statement.

Note	*Frequency*	*Note*	*Frequency*	*Note*	*Frequency*
D#$_1$	38.89	G$_3$	196.00	A#$_5$	932.33
E$_1$	41.20	G#$_3$	207.65	B$_5$	987.77
F$_1$	43.65	A$_3$	220.00	C$_6$	1046.50
F#$_1$	46.25	A#$_3$	233.08	C#$_6$	1108.73
G$_1$	49.00	B$_3$	246.94	D$_6$	1174.66
G#$_1$	51.91	C$_4$	261.63	D#$_6$	1244.51
A$_1$	55.00	C#$_4$	277.18	E$_6$	1328.51
A#$_1$	58.27	D$_4$	293.66	F$_6$	1396.91
B$_1$	61.74	D#$_4$	311.13	F#$_6$	1479.98
C$_2$	65.41	E$_4$	329.63	G$_6$	1567.98
C#$_2$	69.30	F$_4$	349.23	G#$_6$	1661.22
D$_2$	73.42	F#$_4$	369.99	A$_6$	1760.00
D#$_2$	77.78	G$_4$	392.00	A#$_6$	1864.66
E$_2$	82.41	G#$_4$	415.30	B$_6$	1975.53
F$_2$	87.31	A$_4$	440.00	C$_7$	2093.00
F#$_2$	92.50	A#$_4$	466.16	C#$_7$	2217.46
G$_2$	98.00	B$_4$	493.88	D$_7$	2349.32
G#$_2$	103.83	C$_5$	523.25	D#$_7$	2489.02
A$_2$	110.00	C#$_5$	554.37	E$_7$	2637.02
A#$_2$	116.54	D$_5$	587.33	F$_7$	2793.83
B$_2$	123.47	D#$_5$	622.25	F#$_7$	2959.96
C$_3$	130.81	E$_5$	659.26	G$_7$	3135.96
C#$_3$	138.59	F$_5$	698.46	G#$_7$	3322.44
D$_3$	146.83	F#$_5$	739.99	A$_7$	3520.00
D#$_3$	155.56	G$_5$	783.99	A#$_7$	3729.31
E$_3$	164.81	G#$_5$	830.61	B$_7$	3951.07
F$_3$	174.61	A$_5$	880.00	C$_8$	4186.01
F#$_3$	185.00				

FIGURE 11-17.
Musical note frequencies ranging over approximately seven octaves.

The sound effects in the sample programs you've seen so far were created with no particular tune in mind, but you can choose *frequency* values that match musical notes. Because the smallest *frequency* value you can use with the SOUND statement is 37, the lowest note you can play is D#$_1$. (The American Standards Association pitch table actually begins at C$_0$, which has a frequency of 16.35.)

The subscripted number at the end of each note name indicates the octave of that note. If you study Figure 11-17, you'll notice that to raise a note one octave, you simply double its frequency—approximately. For example, C$_2$ has a frequency of 65.41 hertz, and C$_3$ has a frequency of 130.81—almost double the frequency of C$_2$. This fact can help you when you add music to your own programs—all you need to do is double a particular frequency to raise that note one octave.

Practice: Working with the SOUND statement

1. Type in the SOUND-1.BAS program (Figure 11-18).

```
' SOUND-1.BAS
' This program demonstrates the SOUND statement.

CLS

INPUT "Please enter an octave value (1-7):  ", octave%

' play one octave

FOR i% = 1 TO 8
    READ note%                           ' read frequency of note
    note% = note% * (2 ^ octave%)        ' raise note to desired octave
    SOUND note%, 12                      ' play note
NEXT i%

' frequencies of the major notes in the second octave

DATA 65, 73, 82, 87, 98, 110, 123, 131
```

FIGURE 11-18.
SOUND-1.BAS: A program that plays a scale.

2. Run the program and then enter an octave value. Note that the table of frequencies in the DATA statement actually begins at C_2, so if you enter an octave value of 1, the program plays a major scale from C_2 to C_3. And because the SOUND statement accepts only integer values as frequencies, the DATA statement contains values that have been rounded to the nearest integer.

Musical Notation and QBasic

Most of us aren't used to dealing with musical notes in terms of each note's frequency, but QBasic requires that you tell it what notes to play by using their frequencies.

If you plan to add music to your programs—either a tune you wrote yourself or some sheet music you want to copy—you'll have to convert the notes to frequencies. You might want to make a ''cheat sheet'' to help you convert the notes more quickly.

Working from sheet music

Although it's beyond the scope of this book to explain musical notation in any detail, you might find the following tips helpful as you add music to your own programs. Take a look at Figure 11-19, which shows the names of the notes and their positions on the musical staff.

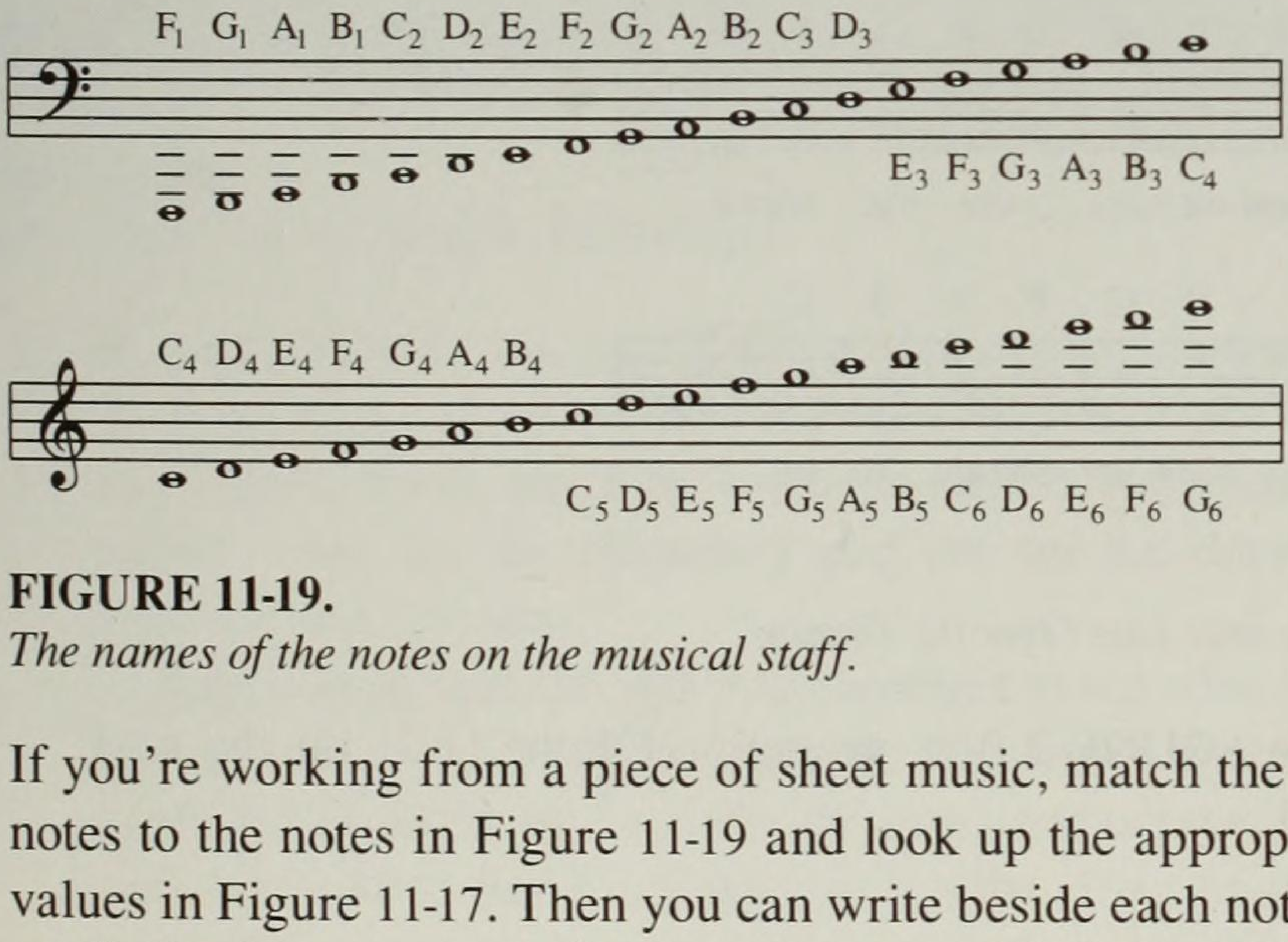

F_1 G_1 A_1 B_1 C_2 D_2 E_2 F_2 G_2 A_2 B_2 C_3 D_3
E_3 F_3 G_3 A_3 B_3 C_4
C_4 D_4 E_4 F_4 G_4 A_4 B_4
C_5 D_5 E_5 F_5 G_5 A_5 B_5 C_6 D_6 E_6 F_6 G_6

FIGURE 11-19.
The names of the notes on the musical staff.

If you're working from a piece of sheet music, match the positions of its notes to the notes in Figure 11-19 and look up the appropriate frequency values in Figure 11-17. Then you can write beside each note on your sheet music the frequency value you need to use with the SOUND statement.

To come up with duration values for your SOUND statements—that is, how long you want each note to last—simply decide how long you want a whole note to be and then divide that value by 2 for half notes, by 4 for quarter notes, and so on. After you've established these values, you can go through your sheet music again and jot down a duration value beside each note.

After you've determined the frequency value and duration value for each note, you're ready to add a melody to your program.

Practice: Playing a song with QBasic

Figure 11-20 shows the music for the song "My Bonnie Lies Over the Ocean." We calculated the frequency value and duration value for each note and then wrote a short program to play the song.

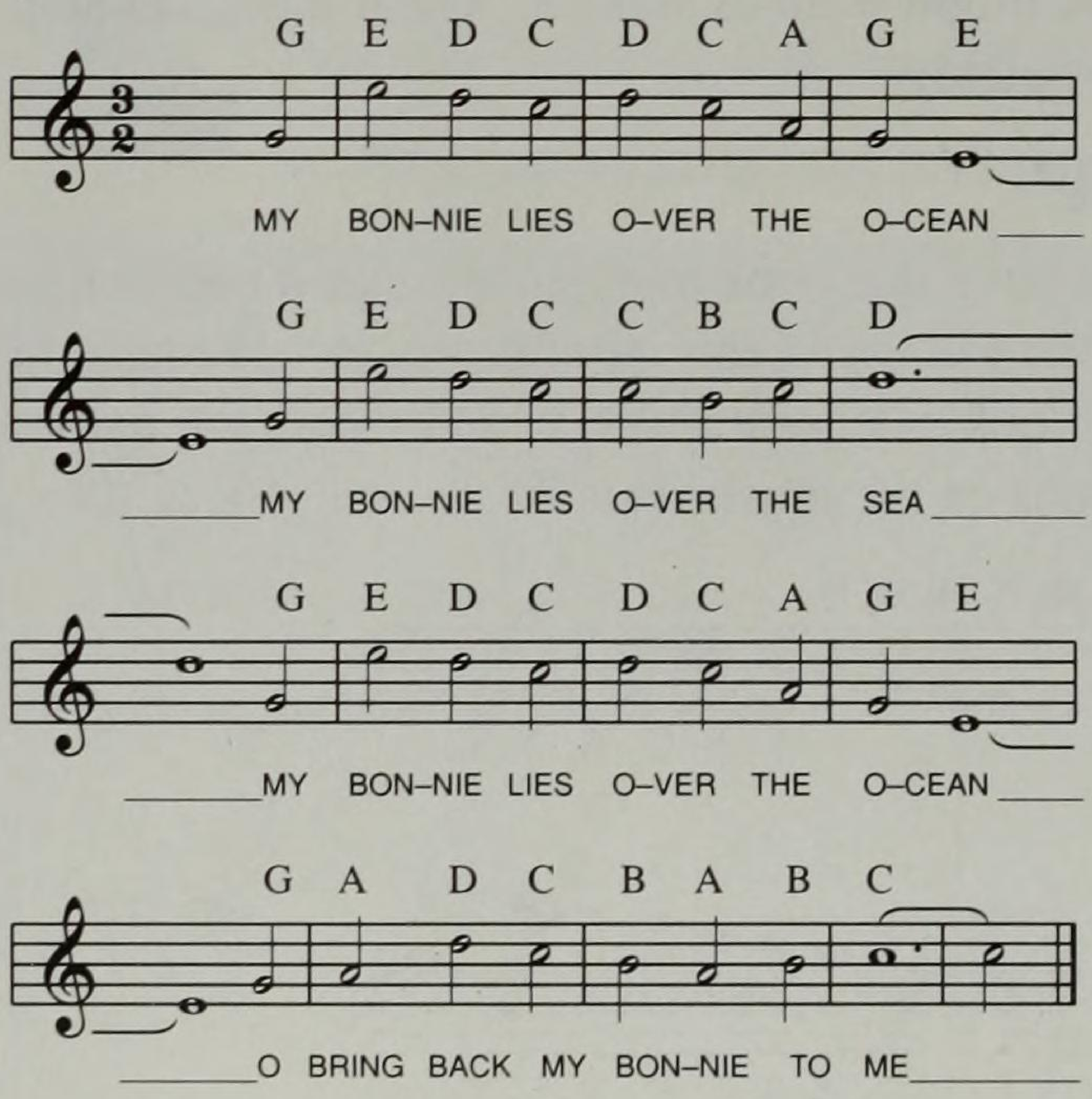

G E D C D C A G E
MY BON–NIE LIES O–VER THE O–CEAN
G E D C C B C D
MY BON–NIE LIES O–VER THE SEA
G E D C D C A G E
MY BON–NIE LIES O–VER THE O–CEAN
G A D C B A B C
O BRING BACK MY BON–NIE TO ME

FIGURE 11-20.
Music for "My Bonnie Lies Over the Ocean."

1. Type in the SOUND-2.BAS program (Figure 11-21 on the next page).

```
' SOUND-2.BAS
' This program demonstrates how to play a song in QBasic.

CLS

INPUT "Press Enter to begin...", dummy$

FOR i% = 1 TO 34
    READ note%, duration%
    SOUND note%, duration%
NEXT i%

'    My       bon-     nie      lies     o-       ver      the
DATA 392, 8,  659, 8,  587, 8,  523, 8,  587, 8,  523, 8,  440, 8

'    o-       cean,    My       bon-     nie      lies
DATA 392, 8,  330, 32, 392, 8,  659, 8,  587, 8,  523, 8

'    o-       ver      the      sea;     My       bon-     nie
DATA 523, 8,  494, 8,  523, 8,  587, 40, 392, 8,  659, 8,  587, 8

'    lies     o-       ver      the      o-       cean--   O
DATA 523, 8,  587, 8,  523, 8,  440, 8,  392, 8,  330, 32, 392, 8

'    bring    back     my       bon-     nie      to       me!
DATA 440, 8,  587, 8,  523, 8,  494, 8,  440, 8,  494, 8,  523, 32
```

FIGURE 11-21.
SOUND-2.BAS: A program that plays a song.

2. Run the program, and you'll hear QBasic play the song. Note how the frequency values and the duration values alternate in the DATA statements. Inside the FOR loop, the READ statement reads two values—one for the frequency and one for the duration—each time through the loop.

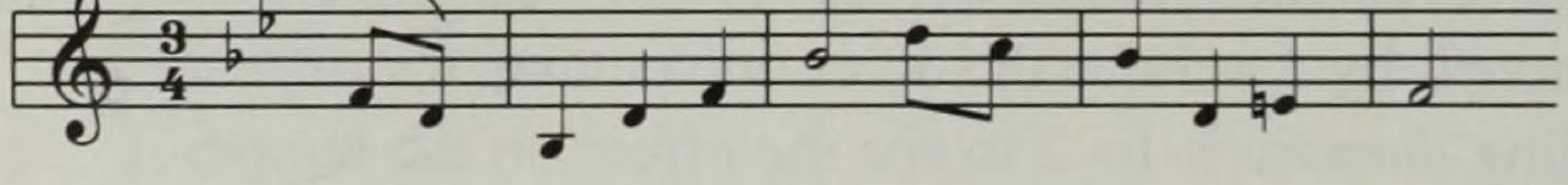
   Experiment with the SOUND statement to see what kind of control you have over the values in the DATA statements. Try multiplying *note%* by 2 to raise the notes one octave, for instance, and try dividing *duration%* by 2 to see the effect on the tempo of the song.

SUMMARY

QBasic contains a wide range of tools you can use to create graphics and sound—and here we've just touched on the basics. After you've completed the exercises in this book, take some time to experiment with the programs you've seen in this chapter. Or use QBasic's on-line help to get more information. Graphics and sound aren't the easiest programming effects to master, but they are certainly two of the most rewarding aspects of QBasic programming.

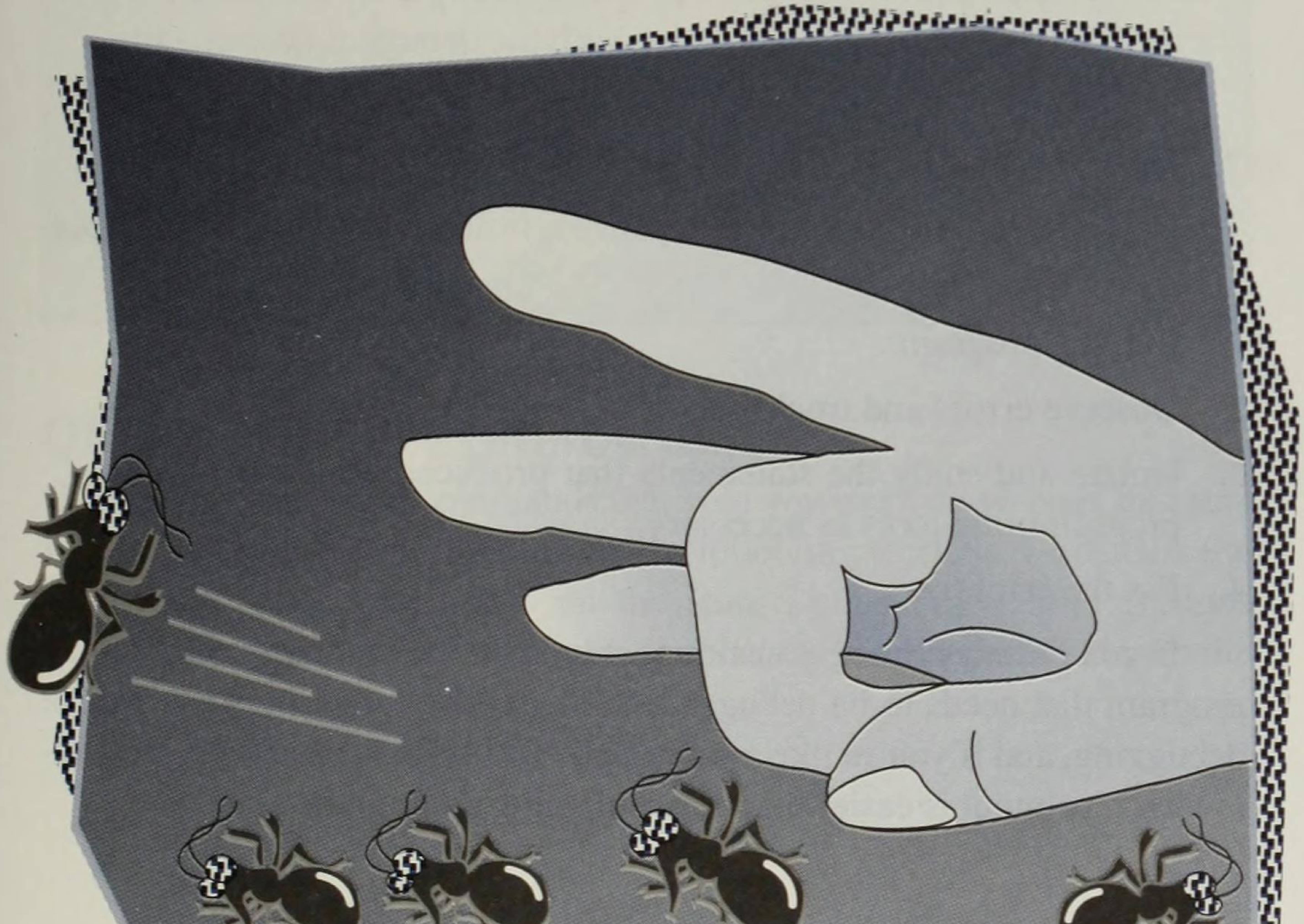
QUESTIONS AND EXERCISES

1. What two components make up your computer's video equipment? What is the function of each component?
2. What is the difference between text mode and graphics mode?
3. What does the LOCATE statement do?
4. Write a short program that asks the user to enter a name and then displays the name at the left edge of the screen and "moves" it to the right edge of the screen.
5. What does the SCREEN statement do?
6. What do the PSET and PRESET statements do, and how do the statements differ?
7. What is the difference between absolute and relative coordinates?
8. True or False: You can use a LINE statement to create a filled box.
9. True or False: You can use a CIRCLE statement to create a filled circle.
10. Write a program that draws a circle near the left edge of the screen and then "moves" it to the right edge of the screen. (Hint: Don't forget to erase the previous circle.)
11. Convert the following tune into a QBasic program:



CHAPTER 12

Debugging QBasic Programs



Remember your first QBasic program? You've come a long way since your output screen displayed *Ward, I'm worried about the Beaver.* Congratulations are definitely in order! But although you're on the road to being a QBasic programming whiz, the day will come when you'll find that a program you've written absolutely refuses to run. At such times you need to have confidence in your ability to diagnose and solve the programming problem. This is where debugging skills come in.

Debugging is the process of tracking down and fixing errors in a program. The ability to debug a problem program is one of the most important skills you can develop as a beginning programmer. Debugging requires you to think modularly—to analyze a program as a computer does, monitoring program execution each step of the way—in order to find the flaw that prevents the program from accomplishing its task.

Fortunately, you're not alone. QBasic provides several powerful debugging commands that help you examine your programs. This chapter introduces these commands and describes how to program to avoid creating bugs in the first place. You'll also debug a sample program from start to finish to get some experience with the debugging commands.

THE PRACTICE OF DEBUGGING

When you debug a program, you repeat the following steps until your program runs successfully:

1. Run the program.
2. Observe errors and trouble spots.
3. Isolate and study the statements that produced each error, using programming tools as necessary.
4. Fix the errors.

If you have a printer, you'll probably want to print out a copy of any program that needs to be debugged. Studying the code is a major part of debugging, and if you're like most people, you'll find that reading and following a printout is easier than scrutinizing a program on screen.

Three Types of Errors

Three types of errors can occur in a QBasic program: *syntax errors, run-time errors,* and *logic errors.*

- A syntax error is a programming mistake that violates the rules of QBasic (such as a missing separator or a misspelled keyword). QBasic points out syntax errors in your program as you type it in and won't let you run your program until each has been fixed.
- A run-time error is a mistake that causes a program to stop unexpectedly during execution. Run-time errors occur when an outside event or an undiscovered syntax error forces a program to stop while it is running. Exceeding the bounds of an array or attempting to open a file with a bad filename are examples of run-time errors.
- A logic error is a human error—a programming mistake that makes the program produce the wrong results. Most debugging efforts are focused on tracking down logic errors introduced by the programmer.

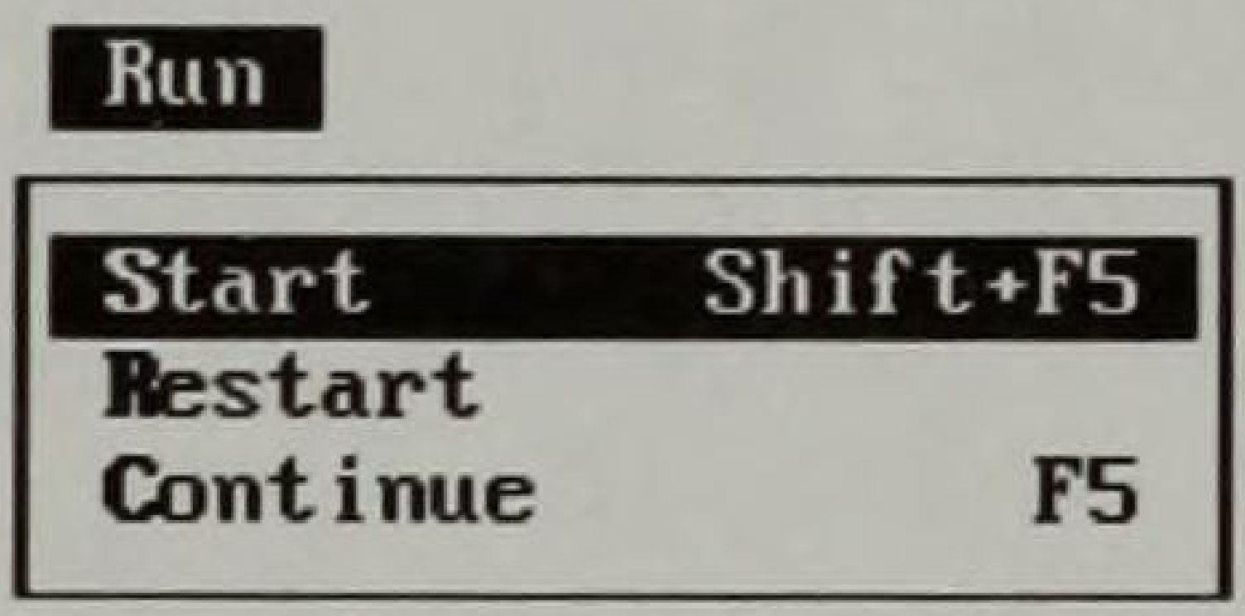
Remember to use your on-line help resources when you encounter error messages produced by syntax errors or run-time errors.

QBASIC MENU COMMANDS

QBasic provides menu commands that help you track down bugs that surface when you run your program. In the following sections, we discuss the debugging-related commands on the Run, Debug, View, and Options menus. For more detailed information about these commands, consult the on-line help.

The Run Menu

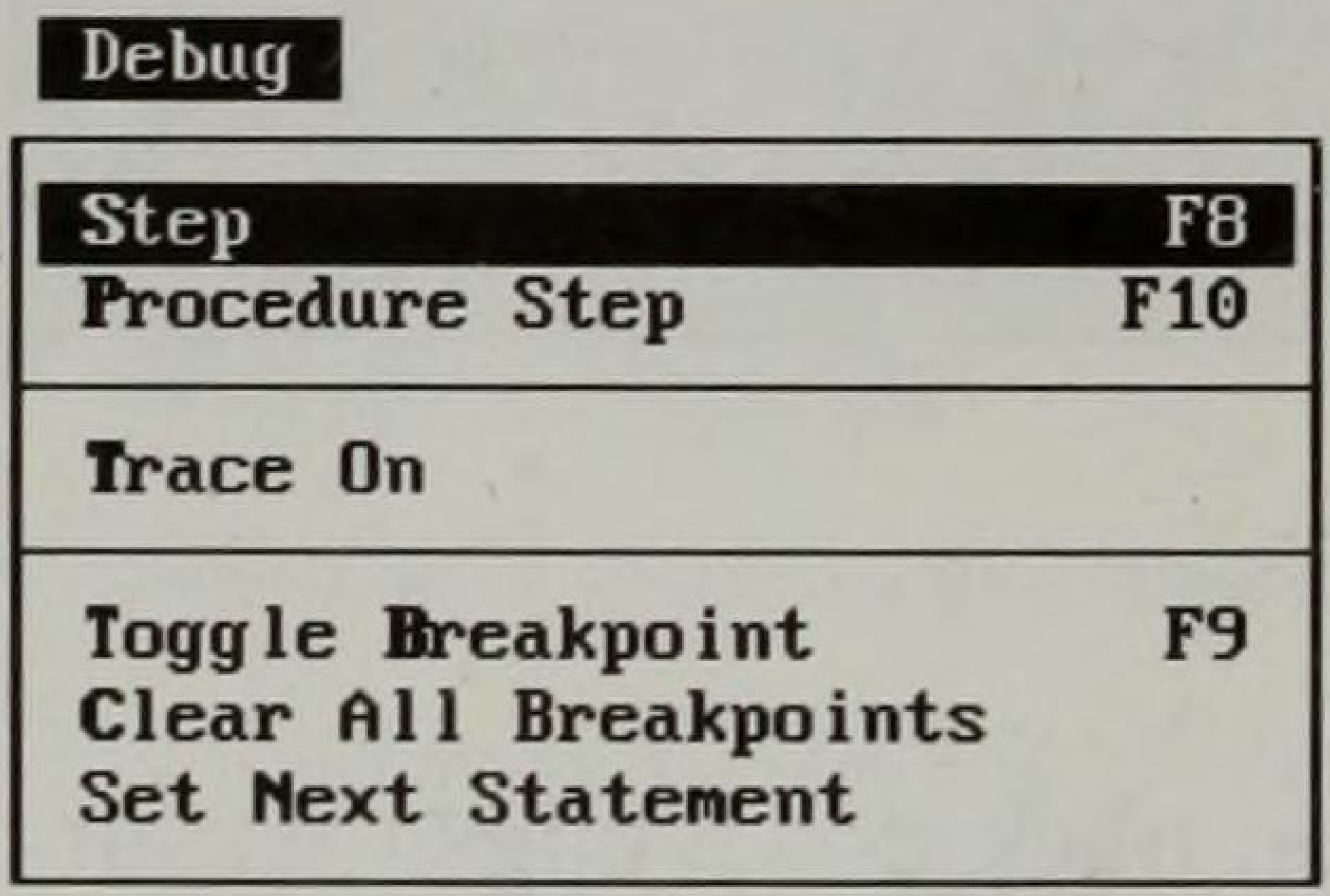
The Run menu contains three debugging-related commands:


Run
Start Shift+F5
Restart
Continue F5

- Start (Shift-F5). Runs the currently loaded program from its beginning at full speed. Press Ctrl-Break to stop a program that is running.
- Restart. Resets all program variables and highlights the first executable statement in your program. Use Restart to move to the top of the program before you execute statements one at a time with the Step and Procedure Step commands on the Debug menu.
- Continue (F5). Runs the currently loaded program from the current line. Use Continue to continue full-speed execution of your program.

The Debug Menu

The Debug menu contains six debugging-related commands:

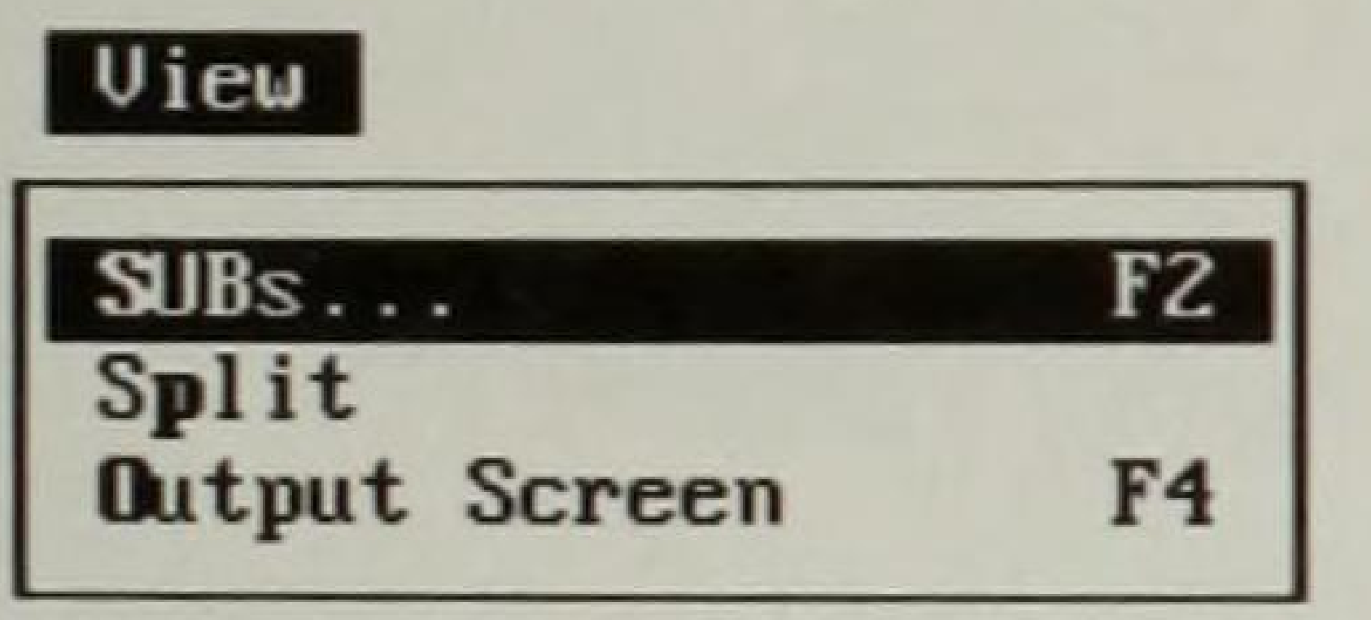

Debug
Step F8
Procedure Step F10
Trace On
Toggle Breakpoint F9
Clear All Breakpoints
Set Next Statement

- Step (F8). Executes one program statement and highlights the next executable statement. Use Step to execute your program one instruction at a time.
- Procedure Step (F10). Executes one program statement and highlights the next executable statement. Note that unlike Step, Procedure Step executes an entire subprogram or user-defined function as one step.

- Trace On. A toggle command that prepares your program for slow-motion execution. When Trace On is active, a program runs slowly and each statement is highlighted as it is executed. When Trace On is inactive (the default), the program runs at its normal speed.
- Toggle Breakpoint (F9). Sets or releases a breakpoint (a line at which program execution will stop) in your program. If you have a color monitor, breakpoint lines appear in red by default.
- Clear All Breakpoints. Releases all breakpoints set by the Toggle Breakpoint command. Use Clear All Breakpoints to release multiple breakpoints at the same time.
- Set Next Statement. Sets the line the cursor is on as the next statement to be executed. Use Set Next Statement to rerun statements or to skip over program statements you don't want to execute.

The View Menu

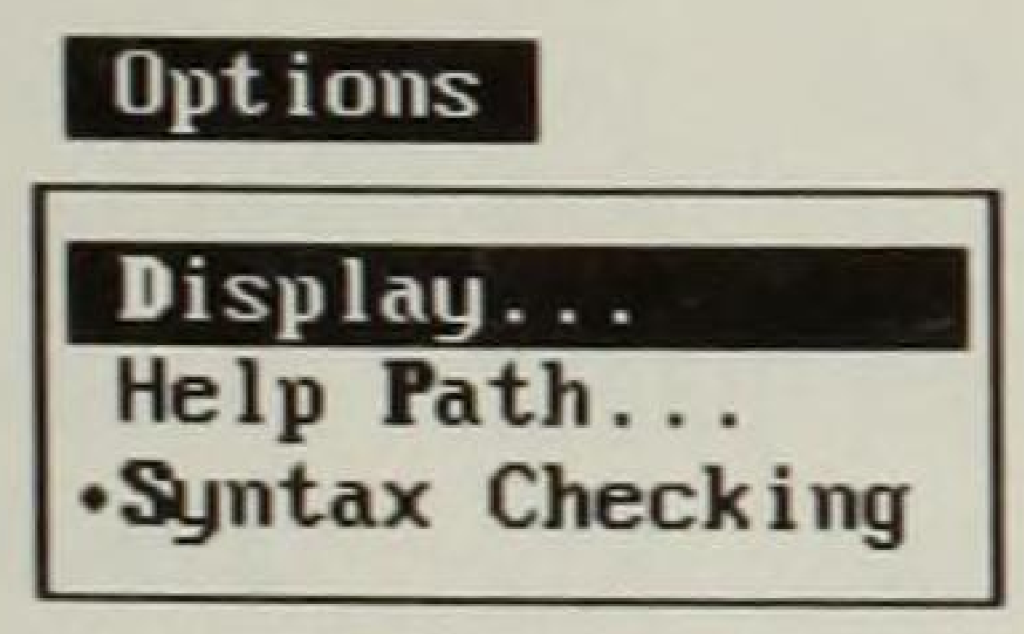
The View menu contains one debugging-related command:



- Output Screen (F4). Displays the contents of the output screen. Use this command while you are stepping through your program, to see what your program displays on the output screen. To return to the editing environment, press any key.

The Options Menu

The Options menu contains one debugging-related command:



- Syntax Checking. A toggle command that turns off QBasic's automatic syntax checker, which is on by default.

In addition to these debugging-related commands, you can use the F7 key to execute the current line and the lines between the current line and the cursor at full speed. Pressing this key is a shortcut for setting a breakpoint with the F9 key and executing from the current line to the breakpoint with the F5 key.

TRACKING VARIABLES WITH PRINT

Another helpful debugging tool is a statement you're already familiar with: the PRINT statement. By placing PRINT statements within a program, or by using them in the Immediate window, you can examine the contents of variables as a program executes. (You'll probably want to display the PRINT statements in a different color to differentiate them from program output.) After you use PRINT statements for debugging, remember to remove them from your program.

The following practice session uses PRINT to find a logic error in a program.

Practice: Using PRINT to track a changing variable

The BUGSHAKE.BAS program (Figure 12-1) is a simple guessing game that asks the user to guess the size in gallons of the largest milkshake ever made (an impressive record set in Ohio in 1988). Although the program runs without syntax errors, it doesn't operate properly: No matter what number the user enters, the program responds that the guess is too low. Can you find the logic error that causes the program to fail?

Type in the BUGSHAKE.BAS program and run it.

```
' BUGSHAKE.BAS
' This program contains a logic error.  Can you find it?

CONST BIGSHAKE% = 174  ' constant equaling gallons in largest shake

CLS

PRINT "How many gallons were in the largest milkshake ever made?"
PRINT

DO WHILE (guess% <> BIGSHAKE%)        ' compare guess to largest shake
    INPUT ; "Guess:  ", guess         ' get guess from user

    COLOR 2                           ' set color to green for clue
    IF (guess% < BIGSHAKE%) THEN      ' determine if guess is high or low
        PRINT , "too low"             '   and print appropriate clue
    ELSEIF (guess% > BIGSHAKE%) THEN
        PRINT , "too high"
    ELSE
        PRINT
    END IF
    COLOR 7                           ' set color back to default white
LOOP

PRINT
PRINT "Correct!  A"; BIGSHAKE%; "gallon strawberry shake was mixed";
PRINT " in Ohio in 1988."
```

FIGURE 12-1.
BUGSHAKE.BAS: A guessing-game program containing a logic error.

You'll receive output similar to the following:

```
How many gallons were in the largest milkshake ever made?

Guess:  100   too low
Guess:  200   too low
Guess:  10000 too low
Guess:  174   too low
Guess:  Ctrl-C
```

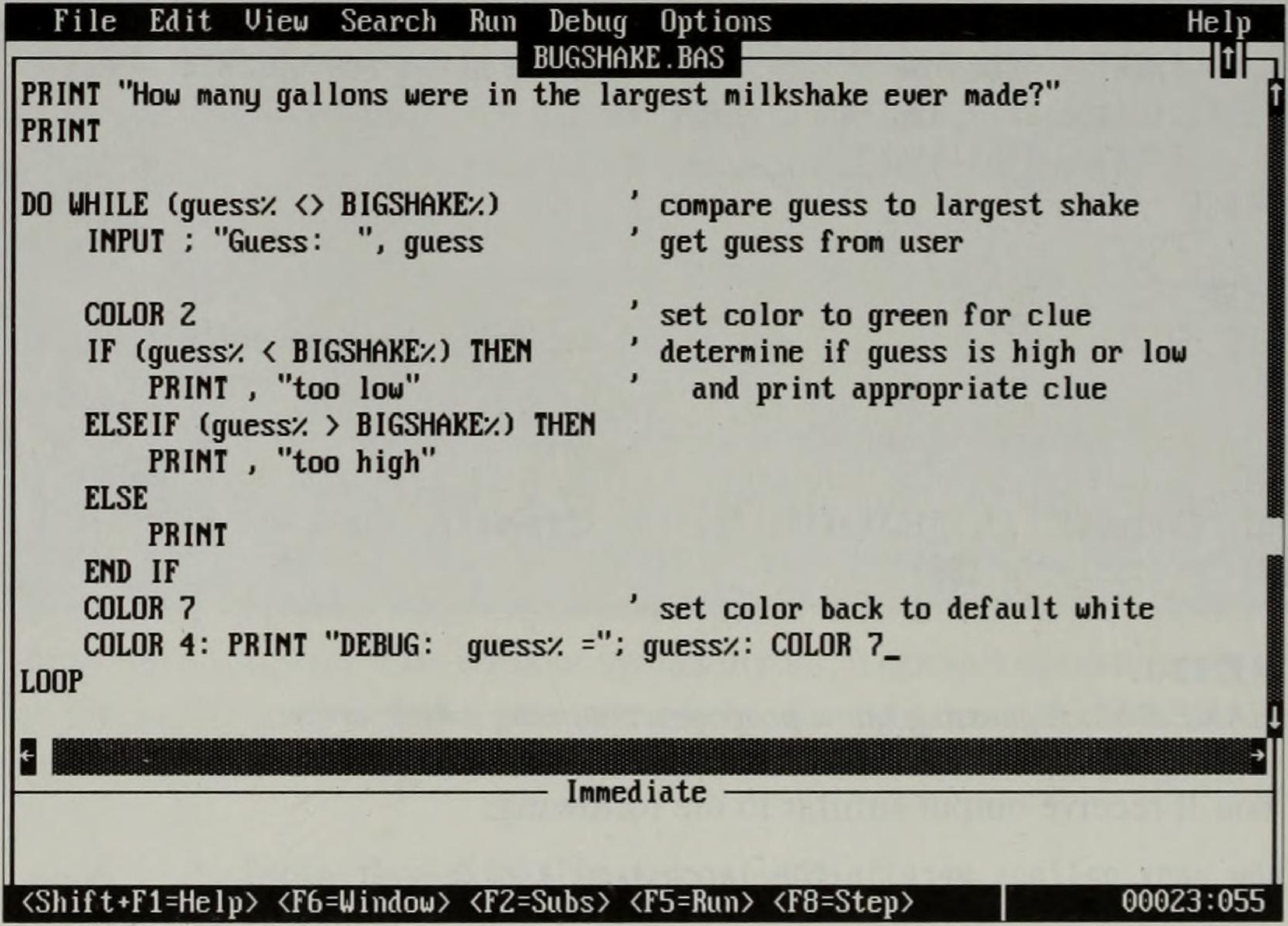
No matter what value is entered (including the correct value, 174), the same message appears: *too low*. The program is stuck in an endless loop. The only way to terminate the program is to press Ctrl-C or Ctrl-Break.

Let's try to track down the error by using a PRINT statement to track the value of the variable *guess%*, which changes each time the user enters a new value in the DO loop.

1. Place the following statements (which display the value of the *guess%* variable in red) at the bottom of the DO loop, just above the LOOP statement:

   ```
   COLOR 4: PRINT "DEBUG:  guess% ="; guess%: COLOR 7
   ```

   When you do so your screen will look like this:


File Edit View Search Run Debug Options Help
BUGSHAKE.BAS
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00023:055

```
PRINT "How many gallons were in the largest milkshake ever made?"
PRINT

DO WHILE (guess% <> BIGSHAKE%)        ' compare guess to largest shake
    INPUT ; "Guess:  ", guess         ' get guess from user

    COLOR 2                           ' set color to green for clue
    IF (guess% < BIGSHAKE%) THEN      ' determine if guess is high or low
        PRINT , "too low"             '   and print appropriate clue
    ELSEIF (guess% > BIGSHAKE%) THEN
        PRINT , "too high"
    ELSE
        PRINT
    END IF
    COLOR 7                           ' set color back to default white
    COLOR 4: PRINT "DEBUG:  guess% ="; guess%: COLOR 7_
LOOP
```

The PRINT statement is surrounded by two COLOR statements: The first changes the foreground color to red, and the second changes it back to white. You might find this second color useful for distinguishing the output of the program from the output of the debugging statements. If your monitor doesn't display color, omit the COLOR statements.

2. Run the program again. You'll receive output similar to the following, complete with debugging information:

```
How many gallons were in the largest milkshake ever made?

Guess:  10    too low
DEBUG:  guess% = 0
Guess:  30000 too low
DEBUG:  guess% = 0
Guess:  174   too low
DEBUG:  guess% = 0
Guess:  Ctrl-C
```

   The result of the PRINT statement is interesting—it shows that for some reason the milkshake guess is not being assigned to the *guess%* variable. There are two explanations for this: Something is wrong with the INPUT statement, or the *guess%* variable has been modified somewhere along the line. Can you find the error?

   If you examine the INPUT statement in the program, you'll discover that the problem is in the *guess%* variable: In the INPUT statement, the *guess%* variable is missing its % type-declaration character:

```
INPUT ; "Guess:  ", guess
```

   As a result, QBasic considers *guess* and *guess%* to be two separate variables.

3. Fix the INPUT statement in the BUGSHAKE.BAS program by adding a % symbol after *guess*:

```
INPUT ; "Guess:  ", guess%
```

4. Run the program again to be sure that it's running properly. If it is, remove the debugging statements you added in Step 1.

Common Programming Errors

The following logic errors pop up from time to time as the result of typing mistakes or design errors. Be on the lookout for them!

- **Incorrect type assignments.** Be sure that data values aren't assigned to variables of the wrong type. Incorrect type assignments can slip by QBasic and cause problems later in the program. Double-check the assignments in each INPUT, INPUT#, and READ statement in your program.
- **Variable-name confusion.** Don't misspell variables or omit type-declaration characters. Also be on the lookout for potential confusion between local and global variables with the same name. Remember: You're on your own when you use variable names—QBasic doesn't check to ensure they're correct.
- **Incorrect logical comparisons.** Check the logical comparisons made by relational operators in IF, SELECT CASE, and looping statements. Use sample programs or the Immediate window to verify that the comparisons work properly.
- **Endless loops.** Beware of loops that can't end. Use the Immediate window to check the termination conditions of your loops.
- **Garbled output.** Misuse of the PRINT, PRINT USING, LOCATE, and SCREEN statements can produce unclear or confusing screen output. Be sure your programs can handle whatever data the user might enter.
- **Array troubles.** Don't confuse an array index number with the value stored in the array. And to avoid the *Subscript out of range* error message, don't read or write past the end of an array.

DEBUGGING A PROGRAM STEP BY STEP

To get some more experience in tracking down and fixing errors, let's debug the DESSERT.BAS program, which uses an array to record your favorite desserts and the number of calories they contain.

A Bug-Free DESSERT.BAS

DESSERT.BAS prompts the user for his or her name and the number of desserts he or she wants to enter. The number the user enters is used to dimension a one-dimensional array of string values in which the dessert names will be stored. A subprogram then adds data to the array and totals the number of calories the user has entered. The program then prints the contents of the array and displays the calorie total in flashing red.

When DESSERT.BAS runs properly, it produces output of the following form:

```
Welcome to the Dessert Program!

Please enter your name:  Mike

How many desserts would you like to enter, Mike? 2

   Dessert name:  Strawberries and cream
   Calories in dessert:  125

   Dessert name:  Chocolate fudge cake
   Calories in dessert:  310

The following is a list of Mike's favorite desserts:

   Strawberries and cream
   Chocolate fudge cake

The list contains a total of 435 calories!
```

Practice: Running the DESSERT.BAS program

But DESSERT.BAS (Figure 12-2 on the next page) contains two logic errors. Type in the DESSERT.BAS program exactly as shown and run it to find out just what's wrong.

```
' DESSERT.BAS
' This program contains two logic errors.  Can you find them?

                                       ' declare GetData subprogram
DECLARE SUB GetData (array$(), num%, totalCalories%)

CLS
                                       ' print welcome message
PRINT "Welcome to the Dessert Program!"
PRINT                                  ' get user's name
INPUT "Please enter your name:  ", userName$
PRINT                                  ' get number of desserts
PRINT "How many desserts would you like to enter, "; userName$;
INPUT num%
PRINT

DIM desserts$(num%)                    ' dimension string array

GetData desserts$(), num%, totalCalories%  ' call subprogram

PRINT "The following is a list of "; userName$; "'s favorite desserts:"
PRINT

FOR i% = 1 TO num%
    PRINT "   "; desserts$(i%)         ' print array contents
NEXT i%

PRINT                                  ' print total number of calories
PRINT "The list contains a total of";  '   with number in flashing red
COLOR 20: PRINT totalCalories%; : COLOR 7
PRINT "calories!"

END

SUB GetData (array$(), num%, totalCalories%)

' The GetData subprogram gets dessert information from the user.
```

FIGURE 12-2. *(continued)*
DESSERT.BAS: A program containing two logic errors.

FIGURE 12-2. *continued*

```
FOR i% = 1 TO num%     ' loop num% times (passed from main program)
   INPUT "  Dessert name:  ", array$(num%)       ' get dessert name
   INPUT "  Calories in dessert:  ", calories%   ' get calories
   PRINT
   totalCalories% = calories%                    ' keep running total
NEXT i%

END SUB                ' return array$ and totalCalories% to main program
```

You'll receive output similar to the following:

```
Welcome to the Dessert Program!

Please enter your name:  Mike

How many desserts would you like to enter, Mike? 2

   Dessert name:  Strawberries and cream
   Calories in dessert:  125

   Dessert name:  Chocolate fudge cake
   Calories in dessert:  310

The following is a list of Mike's favorite desserts:


   Chocolate fudge cake

The list contains a total of 310 calories!
```

Note the two obvious problems: The program lists only one of the values stored in the *desserts$* array, and the program doesn't display the correct calorie total. With these two observations in mind, you can begin to debug the program step by step.

Debugging the DESSERT.BAS Program

The introductory part of the program seems to be OK—it displays helpful information on the screen and obtains data from the user. Let's move past these statements by setting a breakpoint near the middle of the program and running the first section of the program at full speed.

Setting a breakpoint

To set a breakpoint and then run DESSERT.BAS to the breakpoint:

1. Select the Restart command from the Run menu to prepare the program for execution from the beginning. The CLS statement, the first executable statement in the program, will be highlighted.
2. Move the cursor down to the 17th line of the program (the line containing the DIM statement), and select the Toggle Breakpoint command from the Debug menu (F9).
3. Select the Continue command from the Run menu (F5) to run the program to the breakpoint. Enter your own name at the first input prompt and the number *2* at the second input prompt. The program will stop when it reaches the breakpoint.
4. Select the Toggle Breakpoint command (F9) to clear the breakpoint—it's no longer needed.

Stepping through the *GetData* subprogram

Now that we're in bugland, we'll move through the code at a slower pace. Keep your eyes peeled—something that made sense to the person who wrote the *GetData* subprogram isn't working now.

1. Select the Step command from the Debug menu (F8) to execute the DIM statement that dimensions the *desserts$* array. *desserts$* should be large enough to hold all the dessert names the user types in—a number we know in advance and have stored in the *num%* variable. Note that *num%* appears in the DIM statement and that the DIM statement executes without error—we seem to be OK so far.

2. Select the Step command (F8) three times to enter the *GetData* subprogram and execute the first two statements in it. Enter *Cherry pie* at the dessert input prompt.
3. Take a long look at the contents of the *GetData* subprogram to see whether anything looks out of order—sometimes a bug will jump out at you. An excellent way to examine the input process up to this point is to enter a debug PRINT statement in the Immediate window. Let's do this now to see whether the first dessert name is stored properly in the first element of *array$*. Press F6 to move to the Immediate window, and then enter the following line:

   ```
   COLOR 4: PRINT "DEBUG:  array$(1) = "; array$(1): COLOR 7
   ```

   (The colon lets you put more than one statement on a line.) Then the debugging information is displayed on your screen in red:

   ```
   DEBUG:  array$(1) =
   ```

   array$(1), the first element in the *array$* array, should contain the string *Cherry pie*—we just typed it in! But it doesn't, so something must be wrong with the INPUT statement. Specifically, something must be wrong with the index element of the array assignment: The index element should evaluate as 1 the first time through the loop.
4. Press any key to return to the Immediate window.

Isolating the first logic error

We've found the symptom of our first logic error. Let's isolate its cause by taking a close look at the INPUT statement and its role in the FOR loop. A basic rule of thumb for array indexes and loops is that if the loop contains a counter variable and each element of the array needs to be accessed, the counter variable (or some derivative of it) should be used as the array index. The *GetData* subprogram has everything right but the array index: The INPUT statement should be using the counter variable *i%* for the array index, not the *num%* variable.

Fix the error by pressing F6 to return to the View window and then changing the INPUT statement to read as follows:

```
INPUT "   Dessert name:  ", array$(i%)
```

Testing the bug fix

You've corrected the logic error; now let's test the program again:

1. Leave the cursor on the line containing the dessert input line, and select the Set Next Statement command from the Debug menu. The Set Next Statement command lets us rerun the INPUT statement to test our bug fix (saving the time it would take to run the program again from the beginning).
2. Press F8 to execute the new INPUT statement, and then enter *Cherry pie* again. To verify that *Cherry pie* is now stored in the first array element of *array$*, press F6 to move to the Immediate window, move the cursor up to the debug PRINT statement, and press Enter to execute it. This time the correct string value, *Cherry pie*, should appear on the screen. Press any key to return to the Immediate window, and then press F6 to return to the View window.
3. Continue your testing of the FOR loop by moving the cursor down to the line containing the END SUB statement—the last line in the subprogram. Select the Toggle Breakpoint command (F9) to set a breakpoint on this line, and then select the Trace On command from the Debug menu. These two steps let us view the remaining instructions in the FOR loop in slow motion. We specified two desserts at the top of the program, so the loop will be executed twice. Select the Continue command (F5) to start running the program in slow motion. As the loop executes, you'll be prompted to enter dessert data.
4. Looks as if our bug fix did the trick—the program looped and stopped at the breakpoint without error. Select the Toggle Breakpoint command (F9) to remove the breakpoint, and then select the Trace On command again to return execution speed to normal.

Searching for the second bug

One down, one to go. Let's take it slow and search for the last bug in the program:

1. Press F8 once to return to the main program and seven more times to display a few PRINT statements and the contents of the *desserts$* array, which have been passed back to the main program by the *GetData* subprogram.
2. Press F4 to view the current output screen. The correct array elements are all there! Note that the output screen is starting to get a little messy with the debug messages and restarts we've been using. Don't worry too much about this—they won't show up in the final version of our program. Press any key to return to the View window.
3. Press F8 seven more times to execute the statements that display the final calorie count on the screen and end the program. Press F4 to look at the results. The flashing red calorie count is impressive but completely wrong. Let's use a PRINT statement to take a look at the *totalCalories%* variable. Press any key to return to the View window.
4. Move to the Immediate window, change the debug PRINT statement to the following, and press Enter.

   ```
   COLOR 4: PRINT "DEBUG:  totalCalories% ="; totalCalories%: COLOR 7
   ```

   The PRINT statement displays the last calorie count you entered! What's the deal here? Hasn't the *GetData* subprogram been keeping a running total of the calorie counts?

Isolating the second logic error

Let's return to the *GetData* subprogram to see if we can find the source of the error. An easy first thing to check is spelling—perhaps a variable name has been misspelled in the subprogram. Return to the View window, and then use the Find command from the Search menu to search for the *totalCalories%* variable. (The Find command is particularly useful when you're working with large programs.) The Find command locates the *totalCalories%* variable in the parameter list of the *GetData* subprogram.

Select the Repeat Last Find command from the Search menu (F3) several times to find all occurrences of *totalCalories%* in the program. It appears six times—three times in *GetData* and three times in the main program.

Our problem is not a spelling error; these variable names are all spelled the same way. Let's check the statement in the *GetData* subprogram that keeps a running total of the calorie values the user enters. Press F2 and select *GetData* for editing. Examine the assignment statement at the bottom of the FOR loop:

```
totalCalories% = calories%
```

Does this statement properly update the *totalCalories%* variable to keep a running total?

Fixing the second logic error

The answer is *No!* As we've seen throughout this book, in order to update a running total, you need to add the current total *and* the next value added or read and then assign the result to the running total. To fix this assignment statement, change it to read:

```
totalCalories% = totalCalories% + calories%
```

After you've made this change, the program is ready to roll. Test the debugged program by running it from start to finish until you're sure it is completely bug-free.

Avoiding Bugs

Now that you've learned how to track and fix bugs, here are some hints that will help you write bug-free programs in the future.

Plan your program carefully

Before you start to type, be sure you understand what you want your program to do and how you plan to write the program in QBasic. Think about the various algorithms your program will use, how program input and output will be organized, and how data will be stored and manipulated. Start simply, and add complexity as you go.

Work one step at a time

Don't try to write your program all at once. Create and test your program one piece at a time. Isolate different tasks whenever you can by creating subprograms and functions.

Run your program often

When you make a change to your program, run the program to ensure that it still works. By following this rule, you can catch simple programming mistakes early on—before they compound themselves and become major programming problems.

Try new ideas in the Immediate window

Use the Immediate window to test small pieces of program code *before* you place them in the program.

Test your program rigorously

Test your program well each step of the way, and know how your program will respond to any type of input. Consider the following questions:

- What is the biggest number or string this program can handle?
- What is the smallest?
- What will cause this program to "crash"?
- How can I prevent a user from causing this program to crash?
- Will the user understand what this program does?

SUMMARY

In this chapter you encountered a number of debugging tools, tips, and techniques. Debugging tools have come a long way in a few short years, and we're lucky to have such a powerful collection of them right here in QBasic. But as a programmer learning how to debug your own programs (and, worst of all, other people's programs), you need more than debugging tools to help you. Solutions come from thinking logically and creatively about your programs and by examining program execution step by step. The more you know about QBasic and its syntax rules, the better you'll be at detecting program errors and finding solutions for them.

QUESTIONS AND EXERCISES

1. What steps should you follow to isolate a bug in your program?
2. What is the difference between a syntax error and a run-time error?
3. If you were stranded on a desert island with only two commands from the Debug menu, which two commands would you want?
4. What does the F7 function key do?
5. What is the purpose of the Set Next Statement menu command?
6. What type of logic error do you find most difficult to track down and fix?
7. The following program (BADBEAR.BAS) contains two syntax errors. Type in the program, find the errors, and fix them.

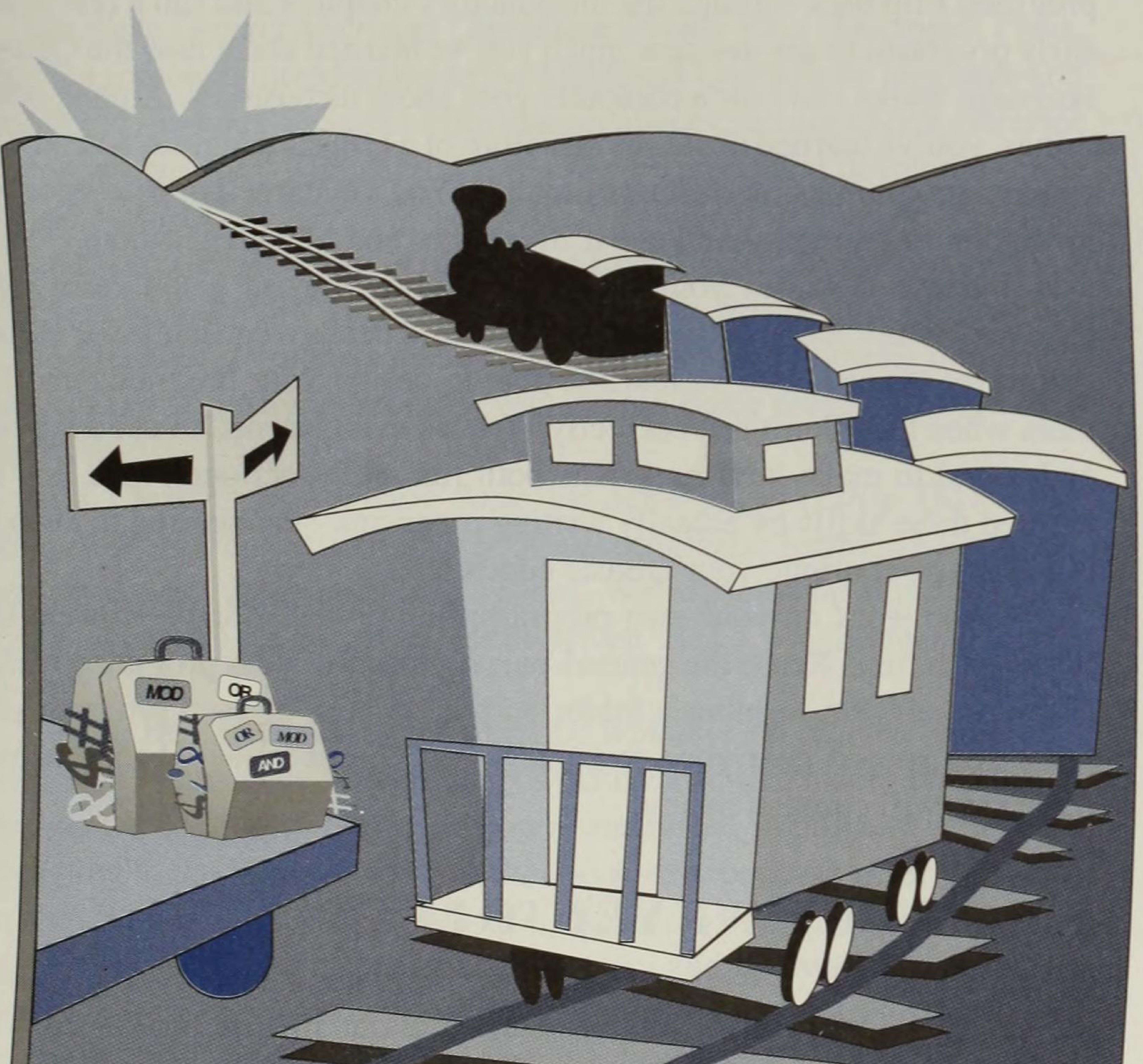
```
' BADBEAR.BAS
' This program contains two syntax errors.  Can you find them?

CLS

DIMM bears$(5)                              ' dimension string array

PRINT "Enter the names of your five favorite bears."
PRINT

FOR i% = 1 TO 5                             ' get 5 strings
   INPUT "Bear:  ", bears$(i%)
NET i%

PRINT
PRINT "You entered the following bears:"
PRINT

FOR i% = 1 TO 5                             ' print 5 strings
    PRINT bears$(i%)
NEXT i%
```

8. The following program (BADNAME.BAS) contains two logic errors. Type in the program and use the debugging tools discussed in this chapter to find and fix the errors.

```
' BADNAME.BAS
' This program separates first and last names and prints them.
'   Can you find the two logic errors?

CLS

PRINT "Enter your first and last name in the following format:  ";
PRINT "Lastname, Firstname"
PRINT

INPUT "Name:  ", fullName$

commaLocation% = INSTR(1, fullName$, ",")

IF (commaLocation% < 0) THEN
    lastName$ = LEFT$(fullName$, commaLocation% - 1)
    firstName$ = RIGHT$(fullName$, LEN(fullName$) - commaLocation% - 1)

    PRINT
    PRINT "What a lovely name!  It's so nice to meet you, ";
    PRINT firstName$; " "; lastName$; "!"
ELSE
    PRINT
    PRINT "Name not in Lastname, Firstname format."
END IF
```

CHAPTER 13

Learning More About Basic


MOD
OR
OR
MOD
AND

Congratulations! You've made it to the end of the line. Although this is the final stop on this particular journey, we hope that it's the jumping-off point for your own exploration of the Basic language. In this chapter we'll briefly review the topics you've covered in this book and introduce you to some books and software that can help you expand and enhance the Basic programming skills you've developed here. In particular, we'll tell you about features of Microsoft Visual Basic for MS-DOS, an advanced version of Microsoft Basic that you can get when you're ready for the next level of programming.

WHAT YOU HAVE LEARNED

You've come a long way in a short time. Take a moment to reflect on your progress: Flip back through the introductory chapters and run a few of the early programs to see just how much you've learned about how the QBasic language works and how a computer goes about its business. Among other things, you've learned about the structure of a QBasic program and how to create a program using menu commands. You've learned how to store information in one place and use it in another and how to get input from a user and use it within your program. You've learned to program efficiently—to use loops for repetitive tasks and declare constants for values that never change. You've learned how to design your programs and fix them when they don't run correctly. You've even learned to use graphics and sound to make your programs both fun *and* functional. And you've learned these skills by actually writing programs—something to keep in mind as you continue your QBasic education.

As you work on your own programming projects, don't reinvent the wheel each time. Reuse the general-purpose parts of your programs whenever you can. Put together a folder with a printout of each program you write and keep it near your computer—you never know when your old programs will come in handy.

WHAT YOU HAVE YET TO LEARN

QBasic's methods of working with data, input and output, and repetitive operations—the methods you've learned about in this book—are the building blocks for each program you write. But there's plenty more to learn about QBasic. You'll likely find that as you continue to work with

QBasic, you'll encounter situations that bring more specific types of questions to mind: How does QBasic work with the mouse attached to my system? How can I access the special capabilities of my printer? How do random-access files work? How can I use QBasic along with other programming languages? The books we describe here can help you find the answers to such questions.

Kris Jamsa's *MS-DOS QBasic,* from the Microsoft Quick Reference Series (Microsoft Press, 1991), is a handy quick-reference guide to every QBasic statement and function.

The Waite Group's MS-DOS QBasic Programmer's Reference (Microsoft Press, 1991) is a comprehensive reference guide to every QBasic keyword and language element. We recommend this book for beginning to advanced QBasic programmers.

The Waite Group's Microsoft QuickBasic Primer Plus (Microsoft Press, 1990) is a thorough introduction to the Microsoft QuickBasic language and compiler. This book includes information about creating Quick libraries and executable files, using serial ports and printers, and developing programs for MS-DOS.

John Clark Craig's *Visual Basic Workshop* (Microsoft Press, 1991) is a collection of over 40 Visual Basic subprograms, forms, and applications for the Microsoft Windows operating system. We recommend this book-disk package as a way to explore event-driven Basic programming under Windows.

John Clark Craig's *Visual Basic for MS-DOS Workshop* (Microsoft Press, 1992) is a collection of over 50 subprograms, forms, and applications for the MS-DOS operating system. We recommend this book-disk package as a way to explore event-driven Basic programming under MS-DOS.

Russell Borland's *Microsoft Word for Windows 2.0 Macros* (Microsoft Press, 1992) introduces users to the powerful Basic-like macro language of Microsoft Word for Windows. If you're interested in pushing your word processing to the limit with Basic, this book-disk package is for you.

Van Wolverton's *Running MS-DOS,* 6th ed. (Microsoft Press, 1993), is a general introduction to personal computers and the MS-DOS operating system. We recommend this book for learning about DOS commands and the operation of the IBM family of personal computers.

OTHER VERSIONS OF BASIC

MS-DOS QBasic is not the only software package for Basic programming available from Microsoft Corporation. In terms of power and features, QBasic actually falls near the low end of the Microsoft family of Basic programming products. As your programming skills grow, you'll want to investigate Microsoft's other Basic language packages:

- *The Microsoft QuickBasic Compiler*
  This full-featured version of MS-DOS QBasic contains more than two dozen additional menu options that support high-level debugging, file and procedure management, and modular programming. The major advantage of the QuickBasic Compiler is that it can create libraries and stand-alone executable files, which brings it to the level of other full-featured programming language products.
- *Microsoft QuickBasic for the Apple Macintosh*
  This QuickBasic compiler runs on the Apple Macintosh family of personal computers. The Macintosh version of QuickBasic is similar to the PC version but has been tailored to support the menu-driven, window-oriented operating environment of the Macintosh.
- *Microsoft Visual Basic*
  Microsoft Visual Basic is Microsoft's newest Basic programming language, available for both the Windows and MS-DOS operating systems. Visual Basic supports most QBasic statements and functions and includes many additional language and design features to make programming easier and faster. Microsoft Visual Basic is the next logical step if you want to continue to program in Basic.

Microsoft Visual Basic

MS-DOS QBasic is a solid and useful product for learning to program in Basic and for developing your own applications. If you want more features, you might consider getting Microsoft Visual Basic for Windows or Microsoft Visual Basic for MS-DOS. Microsoft Visual Basic has a number of advantages over QBasic:

- It allows you to create the user interface of your program with drawing tools.
- It has built-in mouse and printer support.

- It has the ability to create stand-alone executable (.EXE) files for MS-DOS or Windows.
- It has the ability to create libraries of often-used procedures.
- It includes enhanced debugging support.
- It supports multiple-module programs.

Microsoft Visual Basic lets Basic programmers create professional-quality applications quickly and efficiently. And because Visual Basic for MS-DOS and Visual Basic for Windows are compatible, you are able to create applications that can easily be ported from MS-DOS to Windows. Best of all, both Visual Basic products are built around the familiar QBasic language, so the programming skills you've learned in this book will be immediately applicable.

SUMMARY

In this chapter we make it a point to tell you what you have learned. But, more important, we hope that this chapter—and the entire book—will motivate you to explore further, to learn what you have not yet learned. Maybe you picked up this book intending to write the world's greatest program someday. Or perhaps you simply wanted to get a handle on what programming is all about. No matter what your reason, the skills you've learned here will serve you well. Be proud: You've taken the first exploratory steps into a new world. Embrace it! And program, program, program!

APPENDIX A

Using QBasic Menus and Options

This appendix is an at-a-glance reference to the menu-driven QBasic programming environment. It will help you to

- Select menus and commands
- Use dialog boxes
- Use keyboard editing commands
- Select text
- Get on-line help
- Print out your programs or parts of the help system
- Change your display colors
- Use the start-up options

For more extensive help, refer to the *Survival Guide* and *Contents* options of the QBasic on-line help system.

SELECTING MENUS AND COMMANDS

QBasic's eight drop-down menus contain commands for file management, getting help, and editing, executing, and debugging programs. You can select menus and commands by using either the mouse or the keyboard.

With the Mouse…

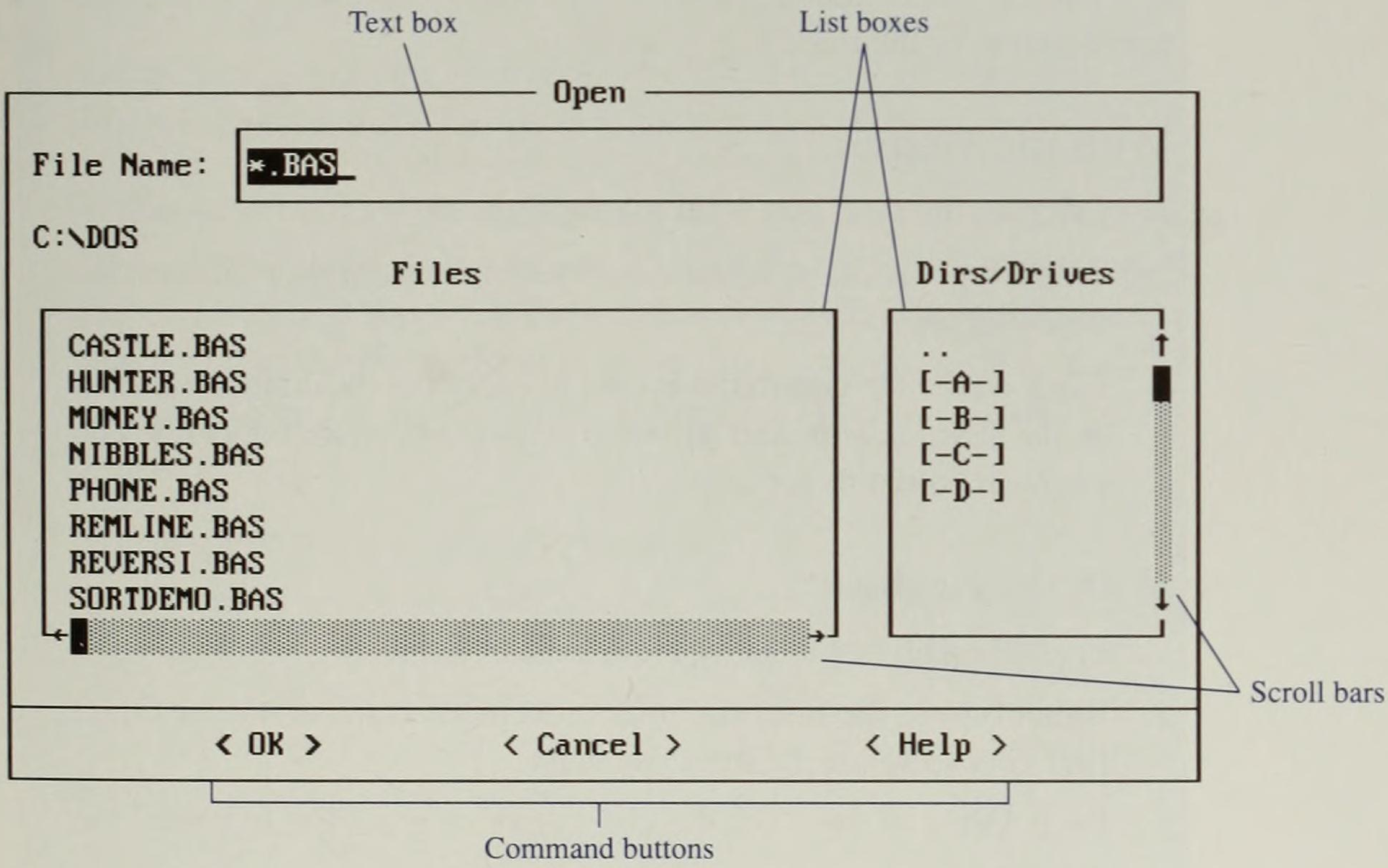
1. Move the mouse pointer directly over the name of the menu you want to select, and click the left mouse button to drop down the menu you want.
2. Move the mouse pointer to the command you want, and click the left button again.

With the Keyboard…

1. Press the Alt key to activate the menu bar.
2. Press the key representing the first letter of the name of the menu you want to select.
3. Use the direction keys to highlight the command you want, and then press the Enter key. (Or simply press the letter that is highlighted in the command name.)

USING DIALOG BOXES

When QBasic needs additional information before it can carry out a command, it displays a *dialog box*. Figure A-1 shows the dialog box that appears when you select the Open command from the File menu. (The contents of your dialog box might be different.)



File Name: *.BAS

C:\DOS

Text box
List boxes
Open
Files
Dirs/Drives
CASTLE.BAS
HUNTER.BAS
MONEY.BAS
NIBBLES.BAS
PHONE.BAS
REMLINE.BAS
REVERSI.BAS
SORTDEMO.BAS
..
[-A-]
[-B-]
[-C-]
[-D-]
Scroll bars
< OK >
< Cancel >
< Help >
Command buttons

FIGURE A-1.
The elements of the Open dialog box.

A typical dialog box contains several fields:

- **Text boxes** receive information you type from the keyboard, such as filenames or search patterns.
- **List boxes** let you select one item (such as a filename, a directory name, or a color) from a number of choices. *Scroll bars* give you access to more information than can fit within a list box.
- **Check boxes** let you indicate whether an element is in effect by the presence or absence of an ''X'' between square brackets.
- **Option buttons** are groups of buttons from which you can make only one selection.
- **Command buttons** control the operation of a dialog box. You use command buttons to carry out a command, cancel a command, or get help.

You can fill in dialog box fields by using the mouse, the keyboard, or a combination of the two.

With the Mouse...

1. Click on the field you want to modify.
2. Either type in the information (in text boxes only) or click on the item you want.
3. Click on the *OK* command button to carry out the command, click on the *Cancel* command button to cancel the command, or click on the *Help* button to get help .

With the Keyboard...

1. Press the Tab key to move from field to field.
2. Either type in the information (in text boxes only) or use the direction keys to select the item you want.
3. Press Enter to carry out the command, or press Esc to cancel the command.

You can use the following keys within a dialog box:

Key(s)	***Purpose(s)***
Tab	Move to next field
Shift-Tab	Move to previous field
Direction keys	Select item in list box; select option button; toggle check box
PgUp/PgDn	Scroll list box several items at a time
Spacebar	Toggle check box; execute command button
F1	Get instructions for filling out dialog box
Enter	Carry out command
Esc	Cancel command

Scroll Bars

Scroll bars let you see more text than is currently visible in a dialog box. Use the scroll bars to bring the hidden text into view.

Using scroll bars with the mouse

To use scroll bars with the mouse, click on the scroll bar arrows to view additional lines or rows of information, and click in the area between the scroll bar arrows to move quickly through a list. You can also drag the scroll box to move to a specific location in the list.

Using scroll bars with the keyboard

To use scroll bars with the keyboard, press the direction keys to view additional lines or rows of information, and press PgUp or PgDn to scroll many lines at a time.

You can also move directly to items starting with a particular letter by pressing the corresponding letter. Continue to press that letter to cycle through all list entries beginning with that letter.

KEYBOARD EQUIVALENTS OF MENU COMMANDS

QBasic provides shortcut keys for most menu commands and for a number of commands that have no menu equivalent. The table below describes the shortcut keys and their uses.

Key(s)	*Purpose*	*Menu/Command*
F1	Display specific help	Help/Topic
Shift-F1	Display general help	Help/Help
F2	Display procedure list	View/SUBs
Shift-F2	Display next procedure	None
Ctrl-F2	Display previous procedure	None
F3	Find next occurrence of search string	Search/Repeat Last Find
F4	Display output screen	View/Output Screen
F5	Continue program execution	Run/Continue
Shift-F5	Execute program from start	Run/Start
F6	Make next window active	None
Shift-F6	Make previous window active	None
F7	Execute program to cursor	None
F8	Execute next statement (step)	Debug/Step
F9	Toggle breakpoint	Debug/Toggle Breakpoint
F10	Execute next statement (step over procedure)	Debug/Procedure Step
Ctrl-F10	Toggle between full screen and multiple windows	None

KEYBOARD COMMANDS FOR EDITING

Along with the editing commands on the Edit menu, QBasic supports a number of keyboard editing commands, most of which have WordStar equivalents. The following table describes the most significant of the editing commands.

Key(s)	*WordStar keys*	*Purpose*
←	Ctrl-S	Move one character left
→	Ctrl-D	Move one character right
Ctrl-←	Ctrl-A	Move one word left
Ctrl-→	Ctrl-F	Move one word right
↑	Ctrl-E	Move one line up
↓	Ctrl-X	Move one line down
Home	Ctrl-QS	Move to first character in line
End	Ctrl-QD	Move to last character in line
Ctrl-Enter	Ctrl-J	Move to beginning of next line
Ctrl-Home	Ctrl-QR	Move to top of procedure
Ctrl-End	Ctrl-QC	Move to bottom of procedure
Ins	Ctrl-V	Toggle insert mode
Shift-Ins	None	Paste contents of Clipboard at cursor
Shift-Del	None	Cut selected text (place in Clipboard)
None	Ctrl-Y	Cut current line (place in Clipboard)
None	Ctrl-QY	Cut from cursor to end of line (place in Clipboard)
Del	Ctrl-G	Delete selected text or character at cursor
None	Ctrl-T	Delete word at cursor
Shift-Tab	None	Delete one indentation level for selected lines
Ctrl-Ins	None	Copy selected text to Clipboard
PgUp	Ctrl-R	Scroll one page up
PgDn	Ctrl-C	Scroll one page down
Ctrl-P	None	Enter control character (type Ctrl-P and then the control character you want)

SELECTING TEXT

Selecting text is the process of highlighting a block of text with the keyboard or the mouse. Selected text can be used in a number of QBasic commands, including the commands shown in the table below.

Menu/Command	*Key(s)*	*Purpose*
Edit/Cut	Shift-Del	Delete selected text (place in Clipboard)
Edit/Copy	Ctrl-Ins	Copy selected text to Clipboard
Edit/Paste	Shift-Ins	Paste contents of Clipboard at cursor
File/Print	None	Print selected text
Search/Find	None	Search for selected text
Search/Change	None	Search for and change selected text
Edit/New SUB	None	Use selected text as name for new subprogram
Edit/New Function	None	Use selected text as name for new function

NOTE: Be cautious with selected text. If you type any character (even a space) while a line of text is selected, whatever you type replaces all the selected text (which is permanently lost). To cancel a selection (to "unselect" it), click the left mouse button or press any direction key.

You can select text by using either the mouse or the keyboard.

With the Mouse…

1. Place the mouse pointer on the first character you want to select. (The "first character" depends on the direction in which you move the mouse—see the following table.)
2. Hold down the left mouse button.
3. Move the mouse pointer to the last character you want to select.
4. Release the mouse button.

Mouse movements have the following effects when you select text:

Movement	*Effect*
Double-click	Select word mouse pointer is on
Right one column	Select character mouse pointer is on
Left one column	Select character to left of mouse pointer
Up one line	Select line mouse pointer is on and line above*
Down one line	Select line mouse pointer is on and line below†

* If the mouse pointer starts in the first column, only the line above the starting point is selected.

† If the mouse pointer starts in the first column, only the line containing the starting point is selected.

With the Keyboard…

1. Use the direction keys to move the cursor to the first character you want to select.
2. Hold down the Shift key.
3. Use the direction keys to move the cursor to the last character you want to select.
4. Release the Shift key.

The direction keys have the following effects when you select text:

Direction key(s)	*Effect*
→	Select character cursor is on
←	Select character to left of cursor
↑	Select line cursor is on and line above*
↓	Select line cursor is on and line below†
PgUp	Select line cursor is on and one screenful up
PgDn	Select line cursor is on and one screenful down
Home	Select from character to left of cursor to beginning of line
End	Select from cursor to end of line
Ctrl-Home	Select from cursor line to beginning of procedure
Ctrl-End	Select from cursor line to end of procedure

* If the cursor starts in the first column, only the line above the starting point is selected.

† If the cursor starts in the first column, only the line containing the starting point is selected.

GETTING HELP

QBasic contains useful on-line documentation for QBasic keywords and language elements, menu commands, error messages, and general programming topics. You can get on-line help in one of three ways:

- For help with your program, place the cursor on the keyword or language element you want documentation for and press the F1 key. If you're using a mouse, click the right mouse button on the keyword or language element.
- For help with a menu command or an error message, press F1 when the menu command is highlighted or when the error message is displayed.
- For help with a general programming topic or with the help system itself, press Alt-H to choose the Help menu, and then select one of the following commands:
  - ☐ *Index* to see an alphabetic listing of QBasic keywords and subjects
  - ☐ *Contents* to see a list of general programming topics including using QBasic, QBasic language elements, and technical information about QBasic
  - ☐ *Topic* to see on-line documentation for the QBasic keyword the cursor is on
  - ☐ *Using Help* to see information about the help system

The QBasic help system is made up of interconnected elements: Each help dialog box contains *hyperlinks,* which take you to other parts of the help system. To select one of these hyperlinks with the mouse, double-click on it. To select a hyperlink with the keyboard, press Tab until the cursor is on the hyperlink you want, and then press Enter.

If the text in the Help window is longer than one screen, use PgUp and PgDn or the scroll bars to view the parts that are off the screen. To switch between the help system, the View window, and the Immediate window, press F6.

When you've finished working with the help system, press Esc to quit.

NOTE: If your computer beeps when you press F1, no on-line help exists for the word your cursor is on.

Copying Programs from the Help System

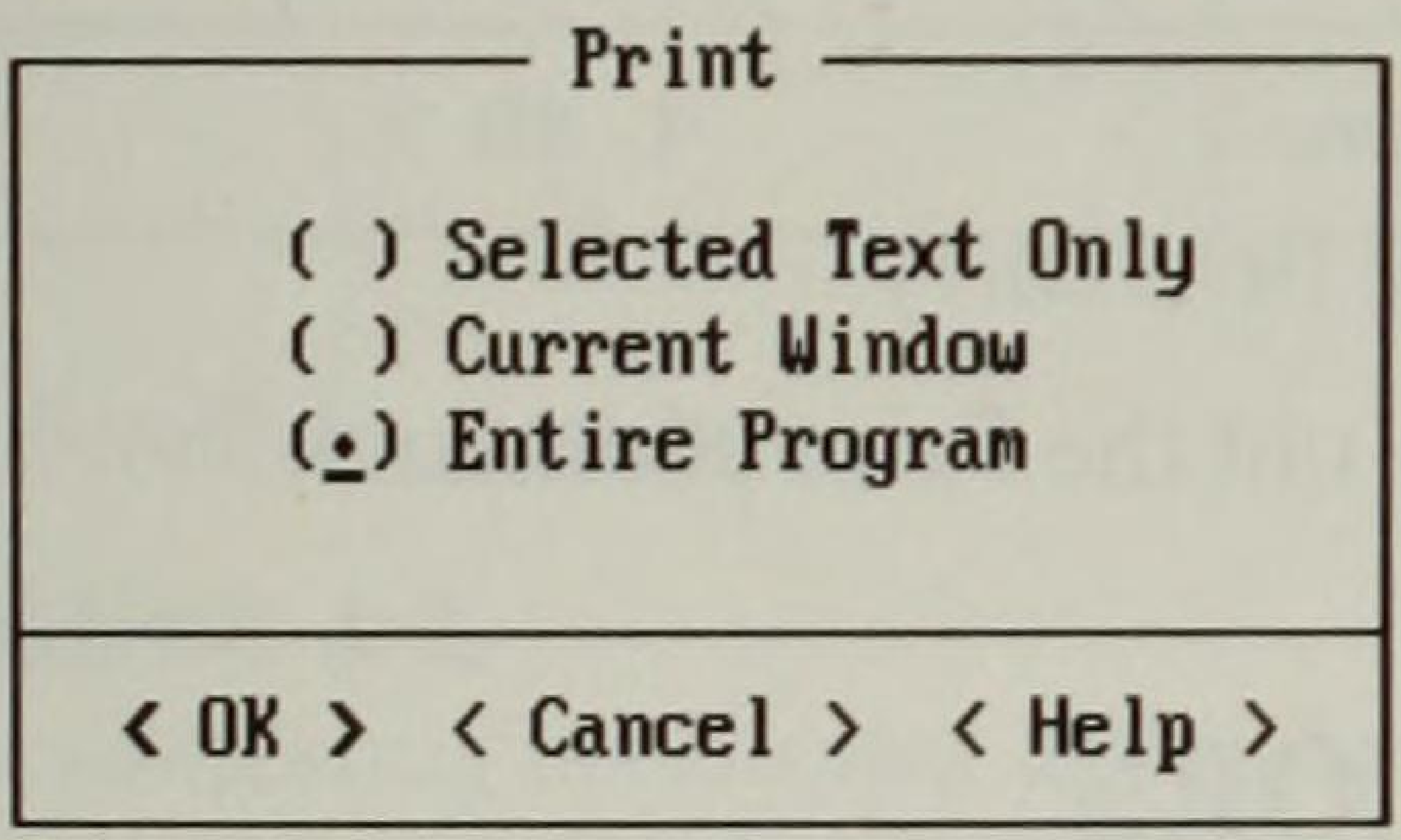
To copy sample code from the help system into your program:

1. Locate the sample code you want to copy, and select it.
2. Select Copy from the Edit menu.
3. Press Esc to close the Help window.
4. Move the cursor to the point at which you want the program code to appear.
5. Select Paste from the Edit menu.

PRINTING

If a printer is attached to your system, you can print out all or part of your program with the Print command on the File menu.

1. Select the Print command. The following dialog box appears:

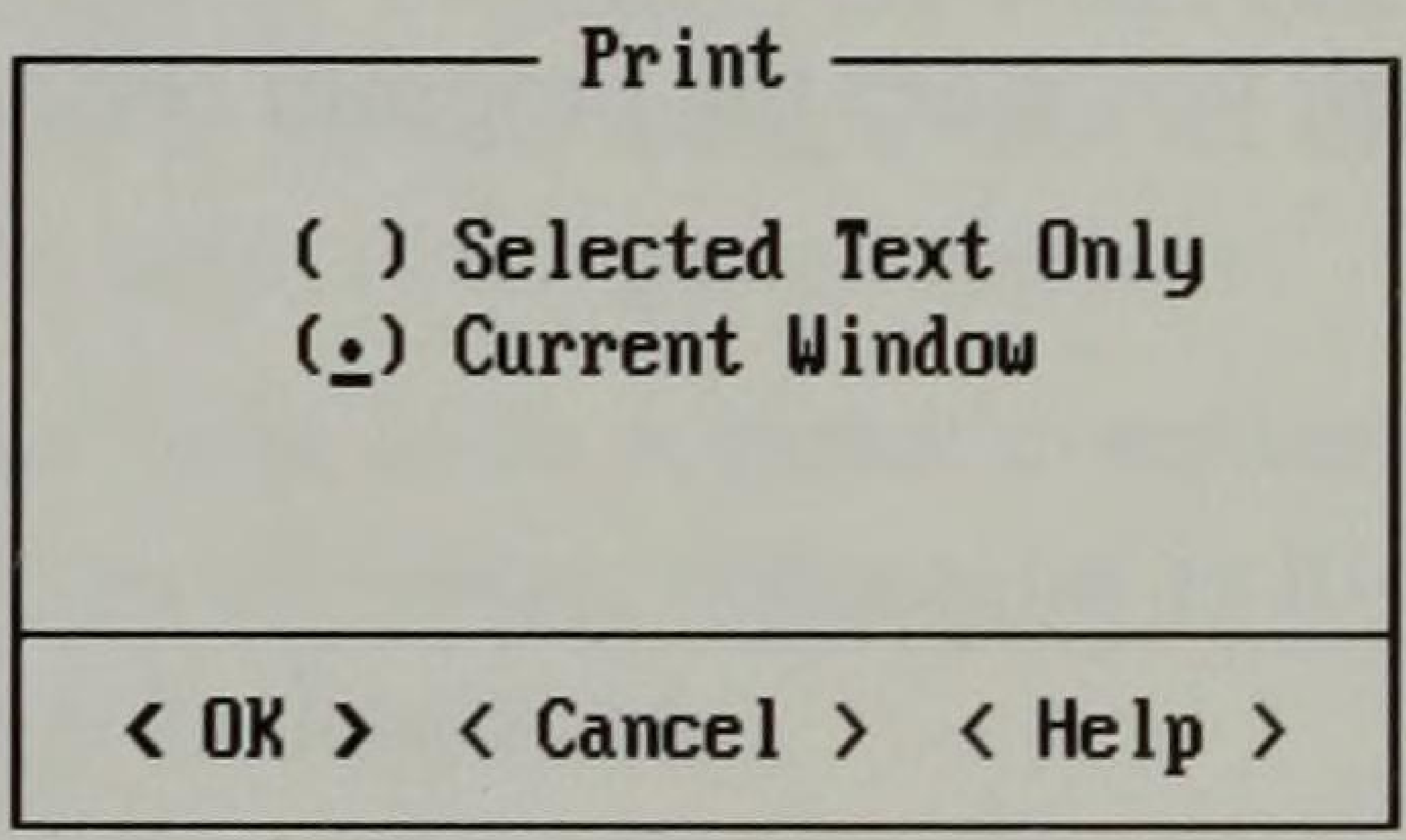


2. Select the appropriate option:
   - *Selected Text Only* prints only the selected text. Before you can use this option, you must select the text in your program that you want to print.
   - *Current Window* prints the contents of the active window only—useful for printing the main program, a procedure, or the contents of the Immediate window.
   - *Entire Program* prints the entire program, including all subprograms and functions.
3. Press Enter or click on *OK* to carry out the Print command.

Printing Part of the Help System

You can also use the Print command to print out selections from the help system.

1. Enter the help system, and locate the entry you want to print.
2. Select the Print command. The following dialog box appears:



Print

( ) Selected Text Only
(•) Current Window

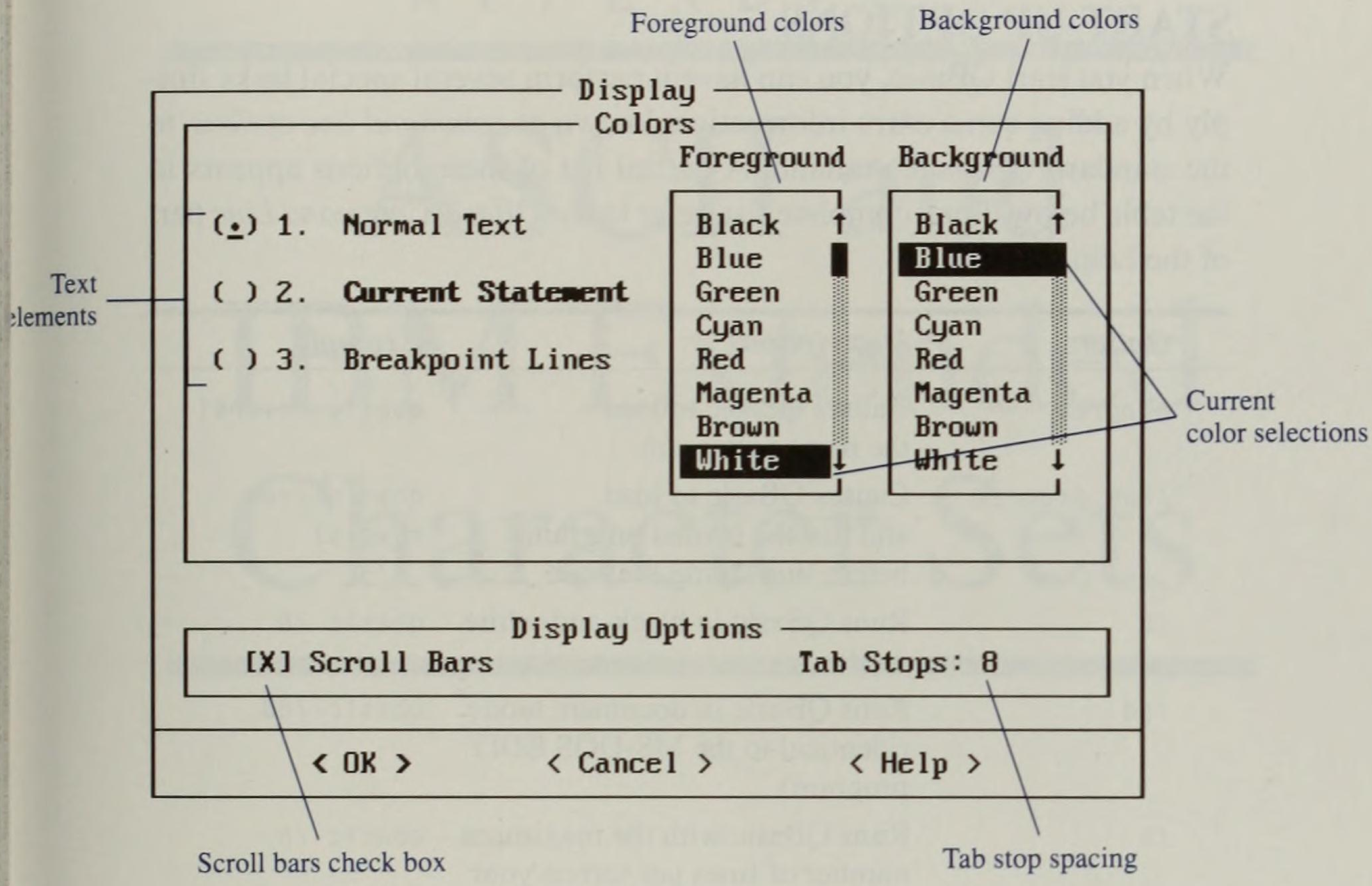
< OK > < Cancel > < Help >

3. Select the appropriate option:
    - *Selected Text Only* prints only the selected text. Before you use this option, you must select the part of the documentation you want to print.
    - *Current Window* prints the entire help entry.
4. Press Enter or click on *OK* to carry out the Print command.

CHANGING DISPLAY COLORS

QBasic lets you change the foreground and background colors on the screen and modify other characteristics of the environment to suit your personal preferences. Your video adapter and monitor dictate which colors you can use.

1. Select the Display command from the Options menu. Figure A-2 shows the Display dialog box that lets you change the foreground and background colors for three text elements on the screen:
    - *Normal text* to change the colors for the regular screen text
    - *Current statement* to change the colors for the next statement to be executed by QBasic
    - *Breakpoint lines* to change the colors for the lines designated as breakpoints by the Debug Toggle Breakpoint command



Display

Colors

Foreground | Background

(•) 1. Normal Text

( ) 2. **Current Statement**

( ) 3. Breakpoint Lines

Foreground: Black, Blue, Green, Cyan, Red, Magenta, Brown, White (selected)

Background: Black, Blue (selected), Green, Cyan, Red, Magenta, Brown, White

Display Options

☒ Scroll Bars

Tab Stops: 8

Foreground colors
Background colors
Text
lements
Current
color selections
Scroll bars check box
Tab stop spacing
< OK >
< Cancel >
< Help >

FIGURE A-2.
The elements of the Display dialog box.

2. Select the text element for which you want to choose colors.
3. Select the colors you want for the foreground and background.
4. Press Enter or click on *OK* to carry out the Display command.

NOTE: If you choose color settings other than the default values supplied by QBasic, resist the temptation to combine bright or contrasting colors. In time, your eyes will thank you for your restraint!

The Display Options part of the dialog box gives you the options to remove the scroll bars from the screen and adjust the spacing of your tab stops. By removing the scroll bars, you can see more of your program at one time; by changing *Tab Stops* to a value less than 8, you can fit more code on the screen and in your printouts (assuming that you use the Tab key to indent your program lines).

START-UP OPTIONS

When you start QBasic, you can have it perform several special tasks simply by adding some extra information, known as *command-line options,* to the standard QBasic command. A partial list of these options appears in the table below. For a complete list, refer to the *QBasic Command Line* part of the help system.

Option	*Description*	*Example*
`program`	Causes QBasic to load the named program	`qbasic reversi`
`/run program`	Causes QBasic to load and run the named program before displaying the code	`qbasic /run reversi`
`/b`	Runs QBasic in black-and-white mode	`qbasic /b`
`/ed`	Runs QBasic in document mode (identical to the MS-DOS EDIT program)	`qbasic /ed`
`/h`	Runs QBasic with the maximum number of lines per screen your hardware can display (typically, 25, 43, or 50)	`qbasic /h`

APPENDIX B

ASCII and IBM Extended Character Sets

Between them, Tables B-1 and B-2 show the 256 characters available on most personal computers that run MS-DOS: Table B-1 shows the ASCII character set, and Table B-2 shows the IBM extended character set. You can display most of these characters by using PRINT and other QBasic statements, and you can compare them by using QBasic relational operators. Chapter 9 describes both the display and compare techniques in detail.

Table B-1: The ASCII character set (codes 0–127)

Character	*ASCII*	*Character*	*ASCII*	*Character*	*ASCII*	*Character*	*ASCII*
	0	<space>	32	@	64	`	96
☺	1	!	33	A	65	a	97
☻	2	"	34	B	66	b	98
♥	3	#	35	C	67	c	99
♦	4	$	36	D	68	d	100
♣	5	%	37	E	69	e	101
♠	6	&	38	F	70	f	102
•	7	'	39	G	71	g	103
◘	8	(	40	H	72	h	104
○	9	)	41	I	73	i	105
◙	10	*	42	J	74	j	106
♂	11	+	43	K	75	k	107
♀	12	,	44	L	76	l	108
♪	13	-	45	M	77	m	109
♫	14	.	46	N	78	n	110
☼	15	/	47	O	79	o	111
►	16	0	48	P	80	p	112
◄	17	1	49	Q	81	q	113
↕	18	2	50	R	82	r	114
‼	19	3	51	S	83	s	115
¶	20	4	52	T	84	t	116
§	21	5	53	U	85	u	117
▬	22	6	54	V	86	v	118
↨	23	7	55	W	87	w	119
↑	24	8	56	X	88	x	120
↓	25	9	57	Y	89	y	121
→	26	:	58	Z	90	z	122
←	27	;	59	[	91	{	123
∟	28	<	60	\	92	¦	124
↔	29	=	61	]	93	}	125
▲	30	>	62	^	94	~	126
▼	31	?	63	_	95	△	127

Table B-2: The IBM extended character set (codes 128–255)

Character	*ASCII*	*Character*	*ASCII*	*Character*	*ASCII*	*Character*	*ASCII*
Ç	128	á	160	@	192	a	224
ü	129	í	161	A	193	b	225
é	130	ó	162	B	194	G	226
â	131	ú	163	C	195	p	227
ä	132	ñ	164	D	196	(	228
à	133	Ñ	165	E	197	s	229
å	134	ª	166	F	198	m	230
ç	135	º	167	G	199	t	231
ê	136	¿	168	H	200	F	232
ë	137	⌐	169	I	201	U	233
è	138	¬	170	J	202	V	234
ï	139	½	171	K	203	d	235
î	140	¼	172	L	204	`	236
ì	141	¡	173	M	205	f	237
Ä	142	«	174	N	206	e	238
Å	143	»	175	O	207	∩	239
É	144	4	176	P	208	[	240
æ	145	7	177	Q	209	6	241
Æ	146	9	178	R	210	≥	242
ô	147	3	179	S	211	≤	243
ö	148	4	180	T	212	⌠	244
ò	149	5	181	U	213	⌡	245
û	150	6	182	V	214	4	246
ù	151	7	183	W	215	'	247
ÿ	152	8	184	X	216	°	248
ö	153	9	185	Y	217	•	249
Ü	154	:	186	Z	218	·	250
¢	155	;	187	[	219	√	251
£	156	<	188	\	220	h	252
¥	157	=	189	]	221	²	253
Pt	158	>	190	^	222	▪	254
ƒ	159	?	191	_	223		255

Some of the ASCII characters (0 through 32, as well as 127) are called *control characters*. You use them to carry out special tasks such as sounding the computer's ''bell'' or advancing the paper in the printer. Table B-3 lists the most frequently used control characters.

Table B-3: Commonly used ASCII control characters

ASCII code	*Control character*	*Description*
7	Bell	Sounds computer's bell
8	Backspace	Backs up one character on screen
9	Tab	Moves right one tab stop on screen
10	Linefeed	Advances printer one line
12	Formfeed	Advances printer one page
13	Carriage return	Advances cursor to next line on screen

APPENDIX C

QBasic Statements and Functions

In this appendix you'll find the name of each statement and function available in QBasic. To see complete documentation for each of these items, including sample programs, start QBasic, type the name of the item, and press the F1 key.

STATEMENTS

- BEEP
- BLOAD
- BSAVE
- CALL
- CALL ABSOLUTE
- CHAIN
- CHDIR
- CIRCLE
- CLEAR
- CLOSE
- CLS
- COLOR
- COM
- COMMON
- CONST
- DATA
- DATE$
- DECLARE
- DEF
- DEFDBL
- DEFINT
- DEFLNG
- DEFSNG
- DEFSTR
- DIM
- DO...LOOP
- DRAW
- END
- ENVIRON
- ERASE
- ERROR
- EXIT
- FIELD
- FILES
- FOR...NEXT
- FUNCTION
- GET
- GOSUB
- GOTO
- IF...THEN...ELSE
- INPUT
- INPUT#
- IOCTL
- KEY
- KILL
- LET
- LINE
- LINE INPUT
- LINE INPUT#
- LOCATE
- LOCK...UNLOCK
- LPRINT
- LPRINT USING
- LSET
- MID$
- MKDIR
- NAME
- ON ERROR
- ON
- OPEN
- OPTION BASE
- OUT
- PAINT
- PALETTE
- PCOPY
- PEN
- PLAY
- PMAP
- POKE
- PRESET
- PRINT
- PRINT USING
- PRINT# USING
- PSET
- PUT

(continued)

STATEMENTS. *continued*

RANDOMIZE
READ
REDIM
REM
RESET
RESTORE
RESUME
RETURN
RMDIR
RSET
RUN
SCREEN
SEEK
SELECT CASE
SHARED
SHELL
SLEEP
SOUND
STATIC
STOP
STRIG
SUB
SWAP
SYSTEM
TIME$
TIMER
TROFF
TRON
TYPE
UNLOCK
VIEW
WAIT
WHILE...WEND
WIDTH
WINDOW
WRITE

FUNCTIONS

ABS
ASC
ATN
CDBL
CHR$
CINT
CLNG
COS
CSNG
CSRLIN
CVD
CVDMBF
CVI
CVL
CVS
CVSMBF
DATE$
ENVIRON$
EOF
ERDEV
ERDEV$
ERL
ERR
EXP
FILEATTR
FIX
FRE
FREEFILE
HEX$
INKEY$
INP
INPUT$
INSTR
INT
IOCTL$
LBOUND
LCASE$
LEFT$
LEN
LOC
LOF
LOG
LPOS
LTRIM$
MID$
MKDMBF$
MKD$
MKI$

(continued)

FUNCTIONS. *continued*

MKL$
MKSMBF$
MKS$
OCT$
PEEK
PEN
POINT
POS
RIGHT$
RND
RTRIM$
SCREEN
SEEK
SGN
SIN
SPACE$
SPC
SQR
STICK
STR$
STRIG
STRING$
TAB
TAN
TIME$
UBOUND
UCASE$
VAL
VARPTR
VARPTR$
VARSEG

APPENDIX D

Converting GW-BASIC Programs to Run Under QBasic

In this appendix, we describe differences between the GW-BASIC programming language supplied with previous releases of MS-DOS and the QBasic programming language supplied with MS-DOS versions 5 and 6, and we give you tips for converting existing GW-BASIC programs to run under MS-DOS QBasic. You need this information if you have GW-BASIC programs you want to run or enhance under QBasic. The good news is that QBasic is quite compatible with GW-BASIC, so most GW-BASIC programs run under QBasic without modification. Sometimes, though, you'll need to modify your GW-BASIC programs to get them to run correctly. In this appendix you'll find the background you need to get started.

PREVIOUS VERSIONS OF BASIC

One fine ''old'' tradition in the personal computer industry is shipping a version of Basic with MS-DOS. MS-DOS distributors recognized early in the life of the personal computer that there would be times when no commercially available program would meet the needs of the computer user and that from time to time people would need to write their own programs. Ever since the original IBM PC was launched in August 1981, Microsoft and IBM and other OEMs (original equipment manufacturers) have shipped one version or another of Basic with MS-DOS.

The limits of the early PC hardware meant that the early versions of Basic shipped with DOS—BASIC, GW-BASIC, BASICA, and other Basics labeled with the names of their manufacturers—had to be compact editions of the language. But over the years, Basic has evolved into a programming environment that boasts the advanced features of professional programming languages and the editing capabilities of powerful word processors. QBasic contains all but a few of the features of the powerful Microsoft QuickBasic Compiler—the bestselling Microsoft language product—yet is still compatible with earlier versions of Basic.

NOTE: Many slightly different implementations of the Basic language have been distributed with microcomputer hardware and software over the years. For simplicity's sake, we'll call all versions of Basic distributed with MS-DOS before MS-DOS 5, GW-BASIC. *Consult your original Basic documentation to see how the version of Basic in your past differs from QBasic.*

What's the Difference?

MS-DOS QBasic differs from GW-BASIC in a number of important ways. The advanced features of QBasic, plus QBasic's compatibility with earlier versions of Basic, make it an extraordinary successor to GW-BASIC.

- **QBasic is a richer programming language.** QBasic contains more statements, functions, and other language elements than GW-BASIC, giving you greater control over your computer and enabling you to create more efficient programs.
- **QBasic has a better editing environment.** QBasic's menu-driven programming environment contains many of the text-processing features found in advanced program editors and word processors. In addition, QBasic supports both keyboard and mouse editing commands and contains a selection of powerful debugging tools.
- **QBasic has online documentation.** The QBasic online help system provides complete language documentation and guidelines for using the QBasic editing environment.
- **QBasic is faster than GW-BASIC.** QBasic's advanced *threaded p-code* interpreter allows QBasic to run Basic programs many times faster than GW-BASIC can. And the programming tools supplied with QBasic make the development process itself (writing, editing, and debugging programs) much faster than in GW-BASIC.

RUNNING GW-BASIC PROGRAMS UNDER QBASIC

To run a GW-BASIC program under QBasic, you save the program in ASCII format, open the program from inside QBasic, and run it. Most GW-BASIC programs run as expected under QBasic without any modification—the only difference you'll notice is that the GW-BASIC program runs a little faster under QBasic.

If QBasic can't run the GW-BASIC program, it displays a *syntax error* message in a dialog box next to the first problem it finds. If this happens, you'll need to modify the GW-BASIC code so that it conforms to the rules of the QBasic language.

Saving the GW-BASIC Program in ASCII Format

All QBasic programs must be in ASCII (plain text) format. By default, the GW-BASIC program editor stores programs in a special compressed format that QBasic can't read, so you need to use the GW-BASIC SAVE command's A option to save the program before you load it into QBasic. To save a GW-BASIC program in ASCII format, follow these steps:

1. Locate a copy of GW-BASIC (or its equivalent) and run it. (MS-DOS versions 5 and 6 do not come with a copy of GW-BASIC—look for GW-BASIC in an earlier version of MS-DOS.)
2. Use the LOAD command to load the GW-BASIC program you want to run.
3. Use the LIST command to verify the contents of the program.
4. Use the SAVE command and the A option to save the program. For example, to save a program named INTEREST.BAS in ASCII format, type the following command in GW-BASIC:

   ```
   save "interest.bas",a
   ```

5. Exit GW-BASIC by typing the following command at the GW-BASIC prompt:

   ```
   system
   ```

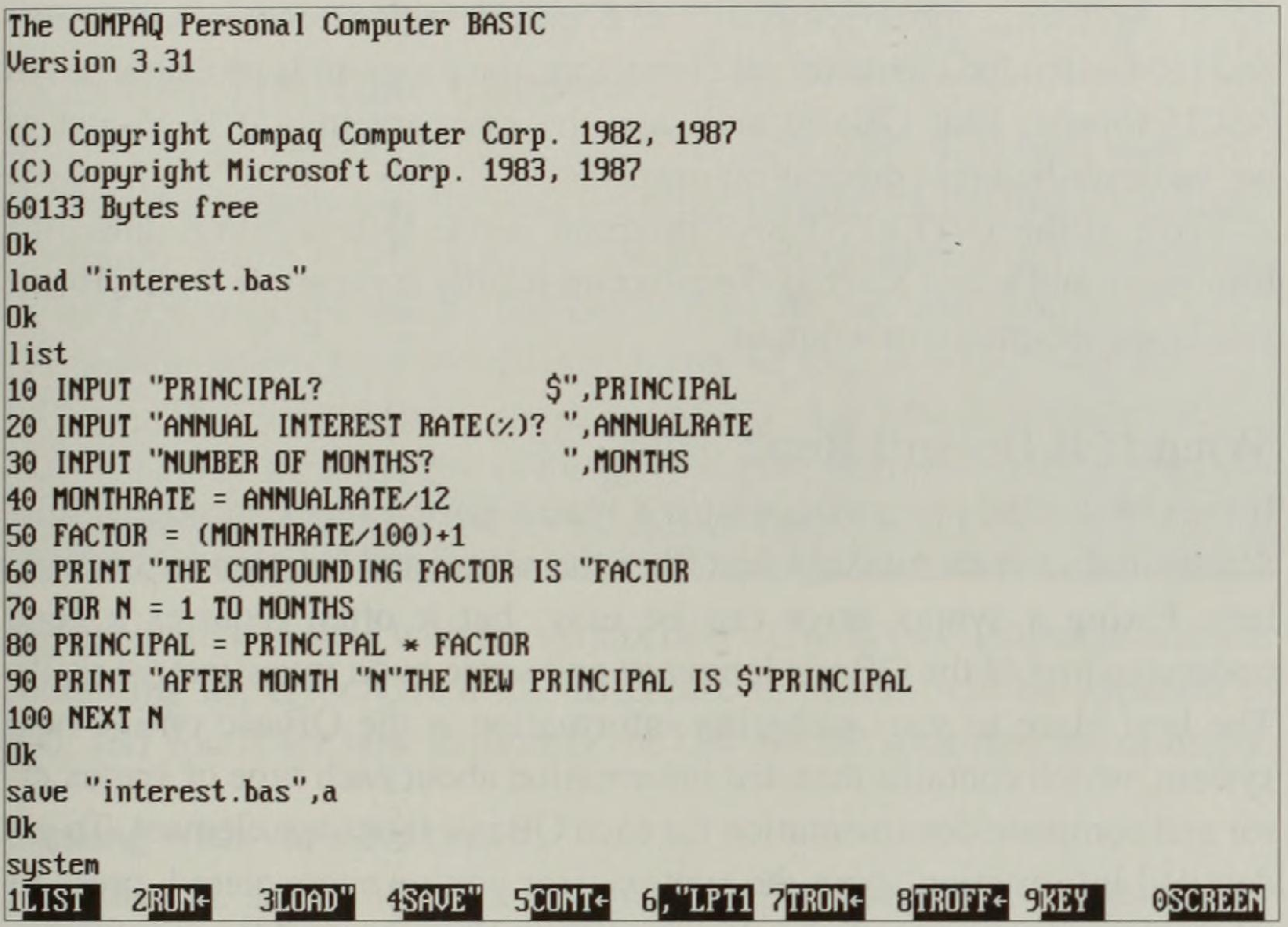
The upper screen on the next page shows the loading, listing, saving, and exiting process in action with the BASICA.EXE and INTEREST.BAS programs that accompanied COMPAQ DOS 3.31.

Running the GW-BASIC Program Under QBasic

After you have saved the GW-BASIC program in ASCII format, you can load it and run it in QBasic. To load the INTEREST.BAS program, for example, type the following at the MS-DOS prompt:

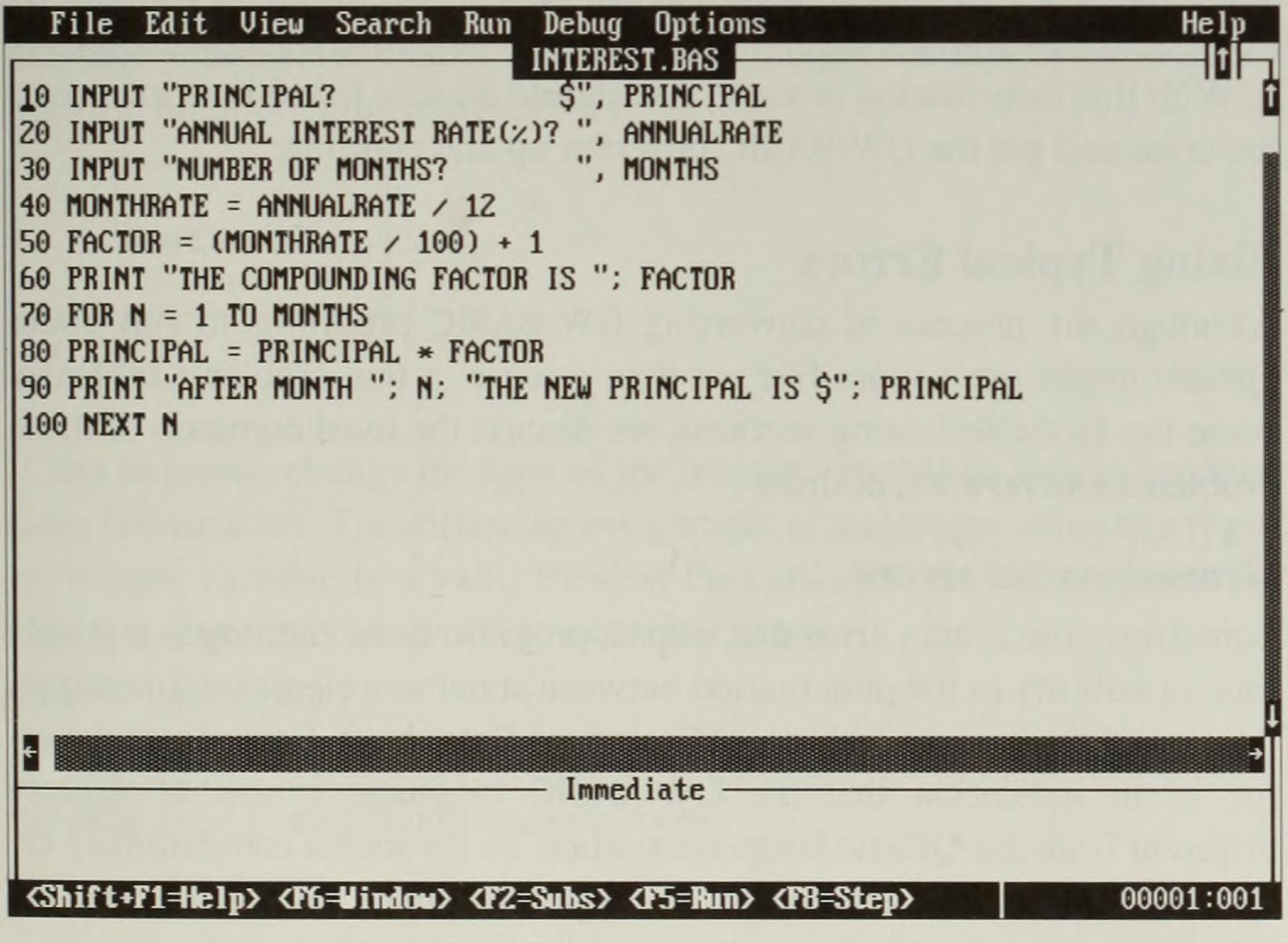
```
C:\>qbasic interest.bas
```

You'll see a screen similar to the lower screen on the opposite page.



```
The COMPAQ Personal Computer BASIC
Version 3.31

(C) Copyright Compaq Computer Corp. 1982, 1987
(C) Copyright Microsoft Corp. 1983, 1987
60133 Bytes free
Ok
load "interest.bas"
Ok
list
10 INPUT "PRINCIPAL?                 $",PRINCIPAL
20 INPUT "ANNUAL INTEREST RATE(%)? ",ANNUALRATE
30 INPUT "NUMBER OF MONTHS?          ",MONTHS
40 MONTHRATE = ANNUALRATE/12
50 FACTOR = (MONTHRATE/100)+1
60 PRINT "THE COMPOUNDING FACTOR IS "FACTOR
70 FOR N = 1 TO MONTHS
80 PRINCIPAL = PRINCIPAL * FACTOR
90 PRINT "AFTER MONTH "N"THE NEW PRINCIPAL IS $"PRINCIPAL
100 NEXT N
Ok
save "interest.bas",a
Ok
system
```

1LIST 2RUN← 3LOAD" 4SAVE" 5CONT← 6,"LPT1 7TRON← 8TROFF← 9KEY 0SCREEN



File Edit View Search Run Debug Options Help
INTEREST.BAS
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00001:001

```
10 INPUT "PRINCIPAL?                 $", PRINCIPAL
20 INPUT "ANNUAL INTEREST RATE(%)? ", ANNUALRATE
30 INPUT "NUMBER OF MONTHS?          ", MONTHS
40 MONTHRATE = ANNUALRATE / 12
50 FACTOR = (MONTHRATE / 100) + 1
60 PRINT "THE COMPOUNDING FACTOR IS "; FACTOR
70 FOR N = 1 TO MONTHS
80 PRINCIPAL = PRINCIPAL * FACTOR
90 PRINT "AFTER MONTH "; N; "THE NEW PRINCIPAL IS $"; PRINCIPAL
100 NEXT N
```

If the listing looks jumbled and contains various control characters and IBM extended character set characters, the program is probably not in ASCII format. Exit QBasic and save the program in ASCII format as we've described that process on page 434.

To run the INTEREST.BAS program inside QBasic, drop down the Run menu and select Start. If the program is fully compatible with QBasic, you'll see the program's output.

What If It Doesn't Run?

If the GW-BASIC program contains a syntax error, execution stops immediately and an error message describes the nature and location of the problem. Fixing a syntax error can be easy, but it often requires a good understanding of the QBasic language and some good investigative skills. The best place to start gathering information is the QBasic online help system, which contains detailed information about each type of syntax error and complete documentation for each QBasic language element. To get detailed information about the syntax error you've encountered, press F1 while the syntax error dialog box is still on the screen. After you understand the nature of the problem, place the cursor on the statement or function associated with the error and press F1 to see online documentation for it. With this information in view, you should be able to see how to correct the error and get the GW-BASIC program up and running.

Fixing Typical Errors

Although the process of converting GW-BASIC programs to run under QBasic might seem more like art than science, a few problems typically come up. In the following sections, we discuss the most common of these problems and how to fix them.

General syntax errors

Sometimes the syntax error that stops a program from running is a simple incompatibility in the punctuation between statement elements (incompatible use of the comma, colon, semicolon, or Enter key). Sometimes the error is an indication that the GW-BASIC language syntax is slightly different from the QBasic language syntax. To fix such a compatibility error, check the online documentation for the language element in question

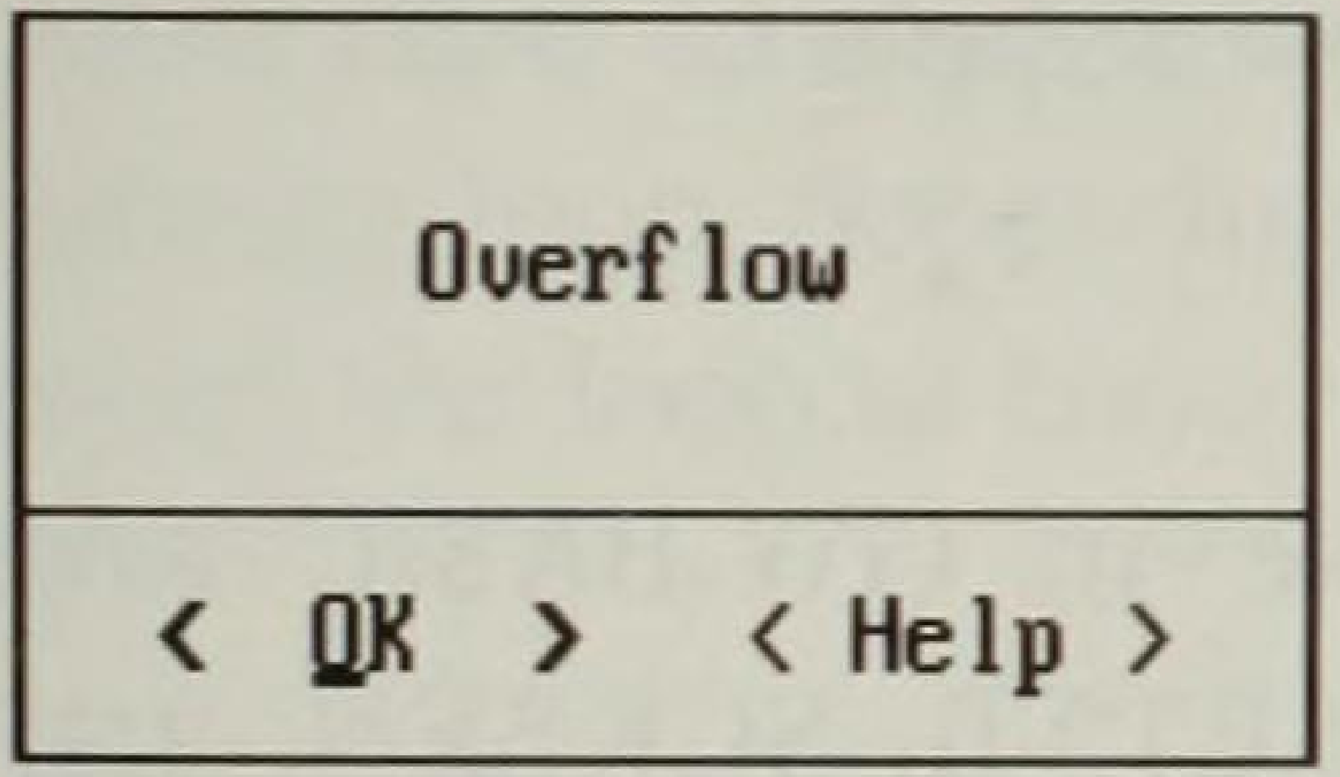
Before You Take Chances

If you didn't write the GW-BASIC program in question (and therefore don't know exactly what the program does on the inside), you might have a little difficulty fixing an incompatibility problem. It's OK to experiment, but before you do, be sure you have a backup copy of the original program so that you can start over again if worse comes to worst. It's also a good idea to spend some time studying a printout of the program to determine exactly what the code does before you try a detailed conversion.

and compare it closely with the syntax of the GW-BASIC language element containing the error. Often the difference in syntax will be apparent to you, and you'll see how to modify the GW-BASIC statement accordingly.

Dealing with variable types

Note that the sizes and types of variables, particularly of integers, vary among versions of Basic. If QBasic has a problem with the way variables are used in a program, you'll see an error message. The following error message appears, for example, if a regular integer variable is too small to hold the number being assigned to it:



Overflow

If this happens, change the type of the integer variable to *long* to hold the extra information. The following assignment of an integer value to a regular integer variable in a valid Hewlett-Packard (HP) BASIC program, for example, generates an overflow error in QBasic:

```
NUM% = 620 * 400
```

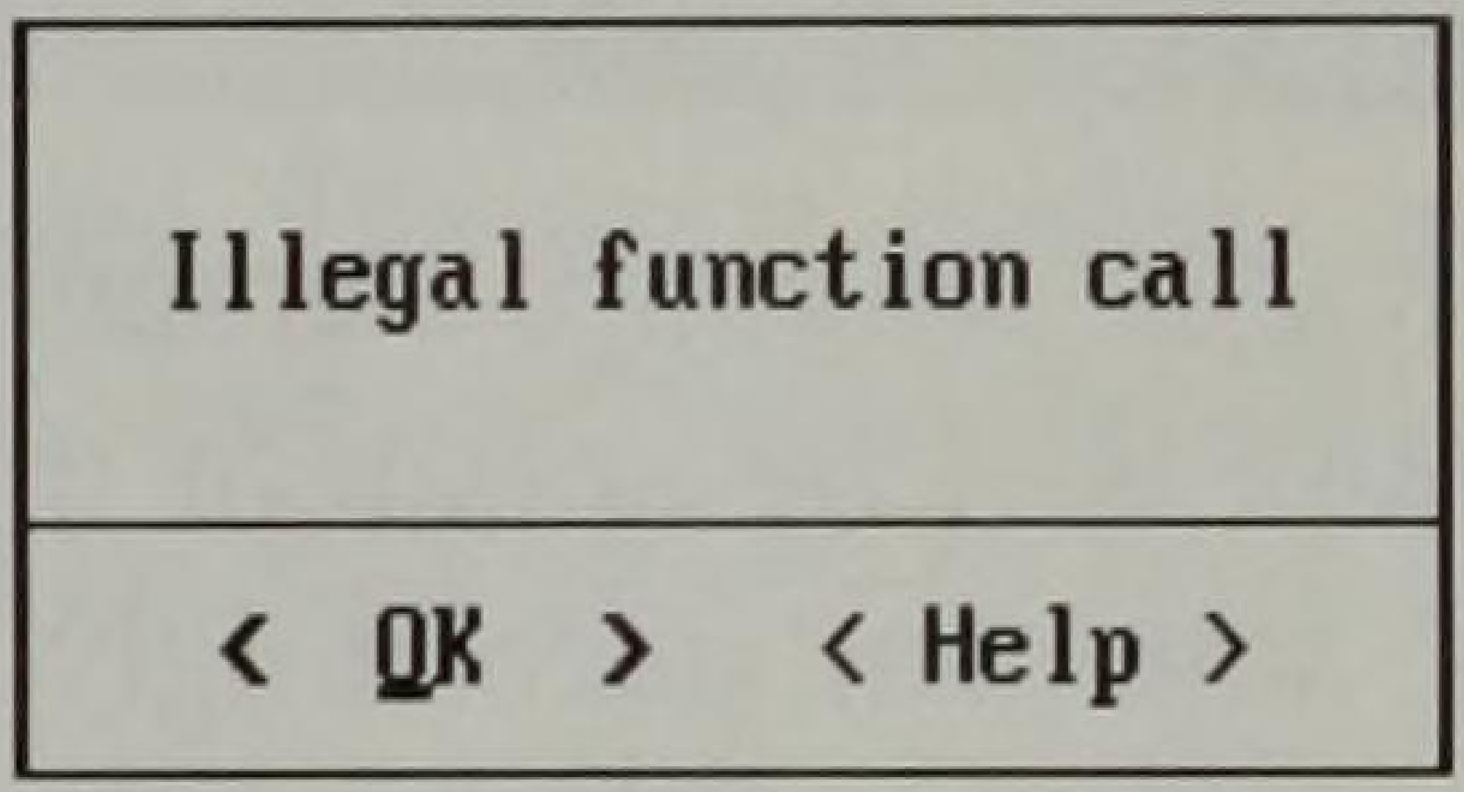
When you convert the program to QBasic, you need to change the variable type to *long* to allow extra room:

```
NUM& = 620 * 400
```

When you change a variable, remember to modify the variable type everywhere in the program—QBasic keeps variables of the same name but different types separate.

Setting the correct screen mode

Many GW-BASIC programs, typically older video games, were designed to support special video modes not supported by QBasic or the video adapter in a typical computer. If you run a program that supports video modes not supported by QBasic or the video adapter in your computer, you'll see the following error message:



```
Illegal function call
```

To fix this problem, you need to make the program support a screen mode supported by QBasic and compatible with the video adapter in your computer. (See Chapter 11, ''Working with Graphics and Sound,'' for more information about video adapters and the SCREEN statement.) Note that if your GW-BASIC program makes heavy use of graphics commands, you might need to convert some of the screen coordinates in your program to match the new screen mode the program will now support.

Getting the right colors

The foreground and background colors available in GW-BASIC and QBasic are controlled in both by the PAINT, PALETTE, SCREEN, and COLOR statements. The ranges and meanings of the arguments to these statements have changed over the years, so some GW-BASIC programs might display different colors when you run them under QBasic. Consult the QBasic online documentation for PAINT, PALETTE, SCREEN, and COLOR to update the program for the color values you want. Note that as computer display hardware continues to evolve, additional color values and screen modes are likely to be added to these statements to give you greater control over the appearance of your program.

Does it run too fast?

A common problem with older GW-BASIC programs, particularly with video games, is that under QBasic the programs can run *too fast.* Speed becomes a problem when an application has been designed to run at a set speed on a certain computer and a faster computer and program interpreter come along and change the running conditions. This is certainly true of some GW-BASIC programs when they run on fast, modern computers under QBasic.

If you need to slow down a program that was written specifically for an older computer and the slower GW-BASIC interpreter, add one or more *timing loops* to the program. A timing loop is a FOR loop that cycles many times in a program without doing anything but passing time—the timing loop simply slows down the action to give the user time to absorb what has happened on the screen. A typical timing loop has the starting value *1* and a large ending value:

```
FOR I% = 1 TO 2000
NEXT I%
```

During the loop nothing happens, but when the loop has finished, enough time has passed to have slowed things down a little.

If you choose to use a timing loop, place it in your program right after the code that displays output on the screen (the output you want the user to have time to absorb). It might take a little tinkering to get just the right timing loop, depending on the speed of your computer and the effect you want the loop to have.

TAKING ADVANTAGE OF QBASIC

Getting your GW-BASIC programs to run without error under QBasic is only the first step in getting the most out of QBasic. After you're up and running, spend some time learning to use the powerful new features of modern, structured QBasic. Not only do these features make QBasic programming easier and faster to learn, but they also make your programs easier to understand and revise later. The following QBasic features (not available in GW-BASIC) are discussed throughout this book. Use them to enhance your programs when the need arises.

- **Extended variable and constant types.** QBasic supports variable names 40 characters long in any combination of uppercase and lowercase characters. QBasic also supports *long integer* variables, *fixed-length string* variables, and numeric and string *constants.* (For more information on these variable types and constants, see Chapter 4, ''QBasic Variables and Operators.'')
- **New decision structures.** Multiline IF statements are new in QBasic and can contain an unlimited number of ELSEIF clauses to help control the path your program takes. The SELECT CASE statement provides a new way of branching conditionally based on the value of a variable. (For more information on these statements, see Chapter 5, ''Controlling Program Flow.'')
- **Improved loops.** In addition to the FOR loop, QBasic supports the DO loop for executing a block of code repeatedly until a certain condition is met. (For more information on the DO loop, see Chapter 6, ''Working with QBasic Loops.'')
- **User-defined subprograms and functions.** QBasic supports multiline blocks of code (with local, independent variables) called *subprograms* and *functions* that can be executed over and over without retyping. User-defined subprograms and functions replace the GOSUB routines and single-line DEF FN functions typically found in GW-BASIC programs. (For more information, see Chapter 7, ''Creating Your Own Subprograms and Functions.'')
- **Style and environment enhancements.** QBasic includes a built-in syntax checker that notifies you of syntax errors as you make them and helps you organize and make your programs easier to read by changing QBasic keywords to uppercase, automatically indenting loops, and enforcing variable name consistency. QBasic doesn't require line numbers (GW-BASIC does), so you can build easier-to-read programs based on carefully planned decision structures, loops, and procedures—without the GOTO and GOSUB ''spaghetti'' code typical of GW-BASIC programs. (For more information, see Chapter 2, ''Getting Started with QBasic.'')

APPENDIX E

Solutions to Questions and Exercises

This appendix contains answers to the questions and exercises at the ends of Chapters 2 through 12.

CHAPTER 2

1. Type *qbasic* at the system prompt.
2. False.
3. The View window contains your program and is the primary window in QBasic. The Immediate window lets you test program lines before you use them in your program.
4. The New command clears the View window and prepares QBasic for a new program. The Open command takes you to the Open dialog box so that you can load an existing program from disk.
5. The View window displays the program you are working on. The output screen displays the output of the program when you run the program.
6. The Cut command deletes a selected block of text from the View window and places it in the Clipboard. The Copy command places a copy of the selected block of text in the Clipboard without removing the original from the View window.
7. You can delete text from a program in the following ways:
   - ☐ With the Cut command on the Edit menu (selected text)
   - ☐ With the Del key (single character or selected text)
   - ☐ With the Backspace key (single character)
   - ☐ With the Ctrl-Y key combination (line)
8. The insert cursor adds text to a line without overwriting the existing characters. The overstrike cursor replaces existing characters with the new characters you type. The visual difference between the two is that the insert cursor appears as a blinking underscore and the overstrike cursor appears as a blinking rectangle. Change back and forth between the two cursors by pressing the Ins key.
9. Eight.

10. The following filenames are invalid:

Invalid filename	*Reason*
PRINTNAME.BAS	Too long (PRINTNAME is nine characters long)
BIG+BAD.BAS	+ character not allowed
BIG RED.BAS	Space not allowed
HI-AND-LO.BAS	Too long (HI-AND-LO is nine characters long)
PROG[1].BAS	*[* and *]* characters not allowed
1ST*PROG.BAS	* wildcard character not allowed

11. Select Exit from the File menu to quit QBasic. If your file contains unsaved changes, QBasic displays a dialog box that contains the following options: *Yes* (save the program), *No* (discard unsaved changes), *Cancel* (return to QBasic), and *Help* (display help).

CHAPTER 3

1. A statement operates in a straightforward manner, usually producing results that are obvious or tangible (a display of characters on the screen, for example). A function generally does its work behind the scenes and returns a value to the program that can be used as an argument to a statement.
2. BEEP — statement
   CLS — statement
   DATE$ — function
   PRINT — statement
   TIME$ — function
3. An item in square brackets is optional. The ¦ character means that you can choose only one of the values in the brackets.
4. An argument is a piece of information supplied to a statement or a function.

5. A string is a collection of characters (which can include letters, numbers, and symbols) surrounded by double quotation marks. A numeric expression is a number or a numeric variable (or, as discussed in Chapter 4, any expression that produces a numeric result).
6. The output of the two PRINT statements is the same.
7. Either a semicolon or a comma at the end of a PRINT statement causes output from the next PRINT statement to be displayed on the same line. A semicolon causes the next PRINT statement's output to appear immediately after the output from the current PRINT statement. A comma causes the next PRINT statement's output to appear in the next print zone.

CHAPTER 4

1. Regular integer, long integer, single-precision floating-point, and double-precision floating-point. They differ in the type and size of number they can hold and in their type-declaration characters.
2. To reserve a space for a minus sign in case the number has been or could become a negative value.
3. An invalid comma likely appeared within a QBasic number.
4. A number outside the range of a variable's numeric data type was assigned to the variable.
5. a. Regular integer
   b. Single-precision floating-point
   c. Single-precision floating-point
   d. Double-precision floating-point
   e. Regular integer or single-precision floating-point
   f. Single-precision floating-point
   g. Long integer
   h. Double-precision floating-point
   i. Single-precision floating-point

6. Regular division returns a fractional result; integer division returns only an integer, discarding any remainder; remainder division returns only a remainder.
7. Exponentiation (^); multiplication and division (*, /, \, MOD); addition and subtraction (+, –).
8. 300.
9. One possible solution to this problem is the CALC.BAS program:

```
' CALC.BAS
' This program calculates and prints four formulas.

CLS

PRINT "ABS(-10) + 5 ="; ABS(-10) + 5
PRINT "SQR(36) ="; SQR(36)
PRINT "SQR(4) ^ 2 ="; SQR(4) ^ 2
PRINT "COS(3.14) ="; COS(3.141592654#)
```

10. One possible solution to this problem is the CIRCLE.BAS program:

```
' CIRCLE.BAS
' This program displays the circumference of a circle when
'   the radius is supplied by the user.

CONST PI# = 3.141592654#             ' declare constant

CLS                                  ' clear screen

PRINT "This program calculates the circumference of a circle ";
PRINT "from its radius."
PRINT                                ' get input from user
INPUT "Enter the radius of the circle:  ", radius!

circum! = 2 * PI# * radius!          ' calculate circumference

PRINT                                ' print result
PRINT "The circumference of the circle is"; circum!
```

CHAPTER 5

1. A numeric expression uses mathematical operators and yields a numeric result. A conditional expression uses conditional operators and yields a true or false result.
2. b, c, f, i, j, k.
3. True.
4. AND requires that both conditions be true before an action is taken; OR requires that only one condition be true before an action is taken.
5. ELSE lets you execute a block of statements when a conditional expression in an IF statement evaluates as false. ELSE is the opposite of THEN.
6. ELSEIF lets you evaluate another condition after a previous IF or ELSEIF statement has evaluated as false. The THEN keyword must appear on the same line as ELSEIF.
7. One possible solution to this problem is the QUESTION.BAS program:

```
' QUESTION.BAS
' This program asks the user a question about programming in Basic.

CLS

INPUT "Do you like programming in Basic so far (Y/N)?  ", reply$
PRINT

IF (reply$ = "Y") OR (reply$ = "y") THEN
    PRINT "Great!  There's more fun to come!"
ELSEIF (reply$ = "N") OR (reply$ = "n") THEN
    PRINT "Sorry to hear that.  Don't worry--it gets better!"
ELSE
    PRINT "Please run the program again."
    PRINT "Enter 'Y' for Yes or 'N' for No at the prompt."
END IF
```

8. QBasic evaluates the expression that appears after SELECT CASE and then checks the following CASE clauses for a matching value. QBasic executes the statements that follow the CASE clause whose values match the expression in the SELECT CASE statement.
9. CASE ELSE lets you specify a block of statements that will be executed if all the CASE conditions in a SELECT CASE statement evaluate as false.
10. IS lets you use a conditional expression in a CASE statement. TO lets you specify a range of numeric values in a CASE statement.
11. One possible solution to this problem is the STATES.BAS program:

```
' STATES.BAS
' This program displays information about one of three states.

CLS

PRINT "Which of the following states would you like to know about?"
PRINT
PRINT "   1) Washington"
PRINT "   2) Virginia"
PRINT "   3) Minnesota"
PRINT
INPUT "State (1-3):  ", reply%
PRINT

SELECT CASE reply%
   CASE 1
      PRINT "** Washington **"
      PRINT "    Population in 1980:  4,538,000"
      PRINT "    Capital:  Olympia"
      PRINT "    Statehood year:  1889"
   CASE 2
      PRINT "** Virginia **"
      PRINT "    Population in 1980:  5,346,818"
      PRINT "    Capital:  Richmond"
      PRINT "    Statehood year:  1788"
```

(continued)

continued

```
    CASE 3
        PRINT "** Minnesota **"
        PRINT "    Population in 1980:  4,075,970"
        PRINT "    Capital:  St. Paul"
        PRINT "    Statehood year:  1858"
    CASE ELSE
        PRINT "Please run the program again and select a number ";
        PRINT "from 1 through 3."
END SELECT
```

CHAPTER 6

1. The counter variable identifies the *value* of the loop—an integer between the *start* and *end* limits of the loop.
2. *start* and *end* can be numeric constants, numeric variables, or numeric expressions. The values can be positive, negative, or zero.
3. 4.
4. The SOUND statement causes your computer's speaker to emit a tone of the specified frequency and duration.
5. A nested loop is a FOR, WHILE, or DO loop inside another FOR, WHILE, or DO loop.
6. Putting the condition at the top of a DO loop ensures that the condition must be met before the loop is executed. Putting the condition at the bottom of a DO loop ensures that the statements in the loop will be executed at least once.
7. Use a FOR loop when you want to execute a block of statements a specific number of times. Use a WHILE or DO loop to execute a block of statements based on the value of a condition.
8. One possible solution to this problem is the GASCASH.BAS program.

```
' GASCASH.BAS
' This program uses a FOR loop to track weekly gasoline expenses.

CLS

PRINT "For each of the seven days of the week, enter the amount ";
PRINT "you spent on gasoline."
PRINT

FOR day% = 1 TO 7
    PRINT "  Cash spent on day"; day%;
    INPUT "-->  $", dayTotal!
    weekTotal! = weekTotal! + dayTotal!
NEXT day%

PRINT
PRINT "Wow!  $"; weekTotal!; "on gas in one week!"
```

9. One possible solution to this problem is the SOUNDER.BAS program:

```
' SOUNDER.BAS
' This program plays a note based on frequency and duration values
'   entered by the user.

CLS

PRINT "Enter frequency and duration values for the sound you want"
PRINT "to hear.  To quit, enter -999 for frequency."
PRINT

DO
    INPUT "Frequency (37-32767):  ", frequency%
    IF (frequency% <> -999) THEN
        INPUT "Duration (0-65535):  ", duration&
        SOUND frequency%, duration&
        PRINT
    END IF
LOOP UNTIL (frequency% = -999)
```

10. One possible solution to this problem is the HIROLLER.BAS program:

```
' HIROLLER.BAS
' This program rolls one simulated die 10 times and displays the
'   message "Nice Roll!" if the die shows 6.

CLS

INPUT "Press Enter to roll the die 10 times.  Think six...", dummy$
PRINT

RANDOMIZE TIMER

FOR i% = 1 TO 10
    roll% = INT(RND * 6) + 1
    PRINT "Roll: "; roll%,
    IF (roll% = 6) THEN
        COLOR 2
        PRINT "Nice Roll!"
        COLOR 7
    ELSE
        PRINT
    END IF
NEXT i%
```

CHAPTER 7

1. All are advantages.
2. The *EnterName$* subprogram name has a string type-declaration character (*$*). This is incorrect—only functions are marked for the type of data they will return. The correct SUB statement would be as follows:

   `SUB EnterName (firstName$, lastName$)`
3. F2.
4. The Procedure Step command on the Debug menu (F10).

5. `COMMON SHARED total%`

6. A subprogram lets you carry out a number of tasks or send a number of values to the main program. A function lets you carry out a smaller, more specific task and returns one value to the main program.

7. The *GetCarFacts* subprogram can be written as follows:

   ```
   SUB GetCarFacts (make$, model$, year%, color$)

   PRINT "Please enter information about your car"
   PRINT
   INPUT "  Car make:   ", make$
   INPUT "  Car model:   ", model$
   INPUT "  Model year:   ", year%
   INPUT "  Car color:   ", color$

   END SUB
   ```

   A statement that calls *GetCarFacts* can be written as follows:

   ```
   GetCarFacts carMake$, carModel$, carYear%, carColor$
   ```

8. The *FindLarger%* function can be written as follows:

   ```
   FUNCTION FindLarger% (int1%, int2%)

   IF (int1% >= int2%) THEN
      FindLarger% = int1%
   ELSE
      FindLarger% = int2%
   END IF

   END FUNCTION
   ```

   A statement that calls *FindLarger%* can be written as follows:

   ```
   PRINT "The larger integer is"; FindLarger%(first%, second%)
   ```

9. One possible solution to this problem is the SHAPES.BAS program.

```
' SHAPES.BAS
' This program displays a shape filled with your favorite character.

' declare constants used as arguments to the COLOR statement

CONST CYAN% = 3
CONST WHITE% = 7

' declare subprograms before they are used; subprogram names and
'   parameters should match those in the subprograms

DECLARE SUB GetShape (symbol$, choice%)
DECLARE SUB PrintLine (char$)
DECLARE SUB PrintRectangle (char$)
DECLARE SUB PrintTriangle (char$)

CLS

' call GetShape subprogram to get desired character and shape

GetShape character$, shape%   ' pass two arguments to GetShape

PRINT
PRINT
COLOR CYAN%

' use CASE statement to call the requested subprogram

SELECT CASE shape%
    CASE 1          ' if shape% = 1, display a triangle
        PrintTriangle character$
    CASE 2          ' if shape% = 2, display a rectangle
        PrintRectangle character$
    CASE 3          ' if shape% = 3, display a line
        PrintLine character$
END SELECT

COLOR WHITE%

END
```

(continued)

continued

```
SUB GetShape (symbol$, choice%)

' The GetShape subprogram prompts the user for a symbol and a shape
'   and returns them to the main program in the symbol$ and choice%
'   variables.

PRINT "This program prints a collection of characters in the ";
PRINT "shape you specify."
PRINT
INPUT "What character would you like to use?  ", symbol$
PRINT
PRINT "What shape would you like to see?"
PRINT
PRINT "    1) Triangle"
PRINT "    2) Rectangle"
PRINT "    3) Line"
PRINT

DO         ' prompt the user until choice% is in the right range
    INPUT "Shape (1, 2, or 3):  ", choice%
LOOP WHILE (choice% < 1) OR (choice% > 3)

END SUB    ' subprogram complete--return to the main program

SUB PrintLine (char$)

' The PrintLine subprogram receives an argument from the main
'   program and uses it to print a line 30 characters long.

CONST LENGTH% = 30     ' set the length of the line at 30 characters

FOR i% = 1 TO LENGTH%  ' display the character 30 times
    PRINT char$;       ' use semicolon to print them one after another
NEXT i%

PRINT

END SUB
```

(continued)

continued

```
SUB PrintRectangle (char$)

' The PrintRectangle subprogram receives an argument from the main
'   program and uses it to print a rectangle 50 characters long by
'   7 characters high.

CONST LENGTH% = 50          ' set length of rectangle at 50 columns
CONST HEIGHT% = 7           ' set height of rectangle at 7 lines

FOR i% = 1 TO HEIGHT%       ' for each of the 7 rows in the rectangle,
    FOR j% = 1 TO LENGTH%   '   display 50 characters one after another
        PRINT char$;
    NEXT j%
    PRINT                   ' print a carriage return after each row
NEXT i%

END SUB

SUB PrintTriangle (char$)

' The PrintTriangle subprogram receives an argument from the main
'   program and uses it to print an equilateral triangle.  The Tab
'   function moves the cursor to the correct column location.

CONST ROWS% = 10      ' set the number of rows to 10
left% = ROWS%         ' use left% to build left side of triangle
right% = ROWS% + 1    ' use right% to build right side of triangle

FOR rowCount% = 1 TO ROWS%            ' for each row in the triangle
    FOR i% = left% TO ROWS%
        PRINT TAB(i%); char$;         ' display left side of row
    NEXT i%

    FOR i% = ROWS% + 1 TO right% - 1  ' display right side of row
        PRINT TAB(i%); char$;
    NEXT i%
```

(continued)

continued

```
    PRINT                  ' print carriage return at end of row
    left% = left% - 1      ' first character in next row will start one
    right% = right% + 1    '   space closer to left margin and extend
NEXT rowCount%             '   one space closer to the right margin

END SUB
```

CHAPTER 8

1. READ statements come first by convention.
2. DATA, READ, and RESTORE work best for data in the following categories:
   - ☐ Data you know about in advance (before the program is run)
   - ☐ Data that always appears in the same order
   - ☐ Data that can be cycled through repeatedly, such as days of the week
3. One possible solution to this problem is the BEAV.BAS program:

```
' BEAV.BAS
' This program reads the names of Beaver Cleaver and his friends from
'   a DATA statement.

CLS

FOR i% = 1 TO 7
    READ pal$
    PRINT pal$
NEXT i%

DATA Beaver, Wally, Lumpy, Whitey, Gus, Eddie, Larry
```

4. False.
5. `DIM numbers!(99)`

6. An end-of-data marker is a number or string indicating that no more data items exist in a list.
7. One possible solution to this problem is the BANZAI.BAS program:

```
' BANZAI.BAS
' This program uses a one-dimensional dynamic string array to store names
'   of characters from the movie "The Adventures of Buckaroo Banzai."

OPTION BASE 1                 ' set array base at 1

CLS

PRINT "** This program collects character names from the film ";
PRINT "'Buckaroo Banzai' **"
PRINT
INPUT "How many names would you like to enter?  ", names%

DIM characters$(names%)       ' dimension array

PRINT                         ' fill dynamic array
FOR i% = 1 TO names%
    INPUT "  Name:  ", characters$(i%)
NEXT i%

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO names%          ' print contents of array
    PRINT characters$(i%)
NEXT i%
```

8. An out-of-range error is a program's attempt to reference an element that does not exist in an array.
9. One possible solution to this problem is the BASEBALL.BAS program.

```
' BASEBALL.BAS
' This program keeps score for a nine-inning baseball game with a
'   two-dimensional array named scoreboard%.

OPTION BASE 1            ' set first array element at 1
DIM scoreboard%(2, 9)    ' dimension 2x9 array for baseball scoreboard

' get team and mascot names

visitor$ = "Boston": visitorMascot$ = "Red Sox"
home$ = "Seattle": homeMascot$ = "Mariners"

CLS

PRINT "Enter runs scored by each team in a nine-inning baseball game."
PRINT

FOR inning% = 1 TO 9     ' get number of runs scored in each inning
    PRINT "Inning"; inning%; "-->    "; visitor$;
    INPUT ; ":  ", scoreboard%(1, inning%)
    PRINT "    "; home$;
    INPUT ":  ", scoreboard%(2, inning%)
                         ' ...and keep running total for each team
    visitorScore% = visitorScore% + scoreboard%(1, inning%)
    homeScore% = homeScore% + scoreboard%(2, inning%)
NEXT inning%

' determine the winner of the game and display results

PRINT
COLOR 2   ' set foreground color to green for "News Flash"

IF (visitorScore% > homeScore%) THEN
    PRINT "News Flash:  "; visitorMascot$; " beat "; homeMascot$;
    PRINT visitorScore%; "to"; homeScore%
ELSEIF (homeScore% > visitorScore%) THEN
    PRINT "News Flash:  "; homeMascot$; " beat "; visitorMascot$;
    PRINT homeScore%; "to"; visitorScore%
```

(continued)

continued

```
ELSE
    PRINT "News Flash:  "; visitorMascot$; " tie "; homeMascot$;
    PRINT visitorScore%; "to"; homeScore%
END IF

COLOR 7   ' set foreground color to white

' display the final scoreboard

PRINT
PRINT "Inning        1    2    3    4    5    6    7    8    9"
PRINT "--------------------------------------------------------"

FOR team% = 1 TO 2        ' for each team in the game
    IF (team% = 1) THEN PRINT visitor$,  ELSE PRINT home$,
    FOR inning% = 1 TO 9  ' ...and for each inning in the game...
        PRINT scoreboard%(team%, inning%); "  ";
    NEXT inning%          ' print the number of runs scored
    PRINT
NEXT team%
```

CHAPTER 9

1. True.
2. False. The correct declaration does not have the *$* type-declaration character:

   `DIM lastName AS STRING * 20`
3. `ONETWOTHREE`
4. a, b, c, d.
5. The value 0 from INSTR means one of the following:
   - ☐ *searchstring* was not found in *basestring*
   - ☐ *start* is greater than the length of *basestring*
   - ☐ *basestring* contains no characters

6. `H`
7. 77.
8. One possible solution to this problem is the GETNAMES.BAS program:

```
' GETNAMES.BAS
' This program gets first and last names from the user and displays
'   them in uppercase.

CLS

INPUT "First name:  ", firstName$
INPUT "Last name:   ", lastName$
PRINT
PRINT UCASE$(lastName$); ", "; UCASE$(firstName$)
```

9. One possible solution to this problem is the REVERSE.BAS program:

```
' REVERSE.BAS
' This program reverses the order of the characters in a string.

CLS
                                         ' get string from user
INPUT "Enter a string of characters to be reversed:  ", inString$
numOfChars% = LEN(inString$)             ' find length of string

FOR i% = numOfChars% TO 1 STEP -1        ' step backwards through string
    tempChar$ = MID$(inString$, i%, 1)   ' extract one letter at a time
    reverse$ = reverse$ + tempChar$      ' build new string
NEXT i%

PRINT                                    ' display new string
PRINT "The characters in reverse order are "; reverse$
```

10. One possible solution to this problem is the DIVIDE.BAS program.

```
' DIVIDE.BAS
' This program divides a string into three parts.

CONST BLANK$ = " "                      ' declare string constant
char$ = ""                              ' initialize variables
charCount% = 1
nameCount% = 0

CLS                                     ' clear screen

PRINT "Enter name in the following format:  First Middle Last"
INPUT "Name:  ", fullName$              ' get 3-part name from user
nameLength% = LEN(fullName$)            ' determine length of name

' loop until the entire three-part string has been stepped through

DO WHILE (charCount% <> nameLength% + 1)

'   read characters one at a time until a blank or end of string
'     is encountered; assign characters to name$ variable

    DO WHILE (char$ <> BLANK$) AND (charCount% <> (nameLength% + 1))
        char$ = MID$(fullName$, charCount%, 1)
        name$ = name$ + char$
        charCount% = charCount% + 1     ' track number of character read
    LOOP

    char$ = ""                          ' reset char$
    nameCount% = nameCount% + 1         ' increment nameCount%

    SELECT CASE nameCount%              ' assign string to name variables
        CASE 1                          '   based on value of nameCount%
            firstName$ = name$
        CASE 2
            middleName$ = name$
        CASE 3
            lastName$ = name$
    END SELECT
    name$ = ""                          ' reset name$
LOOP
```

(continued)

continued

```
PRINT
PRINT "Results of separation process:"
PRINT
PRINT "First name is "; firstName$    ' display results
PRINT "Middle name is "; middleName$
PRINT "Last name is "; lastName$
```

CHAPTER 10

1. OUTPUT deletes the contents of the existing file; APPEND adds to the end of the existing file.
2. d.
3. False.
4. LINE INPUT# is more useful than INPUT# when you need to read long lines of string data or lines containing commas from a file.
5. A literal argument to the SHELL statement must be enclosed in quotation marks. The correct statement would be as follows:

   `SHELL "copy test.txt test2.txt"`
6. `Search string found!`
7. One possible solution to this problem is the CITY.BAS program:

```
' CITY.BAS
' This program stores a list of cities in a sequential file.

OPEN "CITY.TXT" FOR OUTPUT AS #1      ' open file in current drive/dir

CLS

PRINT "This program stores city names on disk in a file named CITY.TXT."
PRINT "Enter your favorite cities and type END to quit."
PRINT
```

(continued)

continued

```
DO WHILE (city$ <> "END")             ' until user enters END
    INPUT "  City name:  ", city$     '   get names from user
    IF (city$ <> "END") THEN PRINT #1, city$   ' write names to file
LOOP

CLOSE #1                              ' close file

PRINT
INPUT "Press Enter to see the cities you entered.  ", dummy$
PRINT

OPEN "CITY.TXT" FOR INPUT AS #1       ' open file for input

DO WHILE (NOT EOF(1))                 ' until end of file is reached
    INPUT #1, city$                   '   get names
    PRINT city$                       '   print names
LOOP

CLOSE #1                              ' close file
```

8. One possible solution to this problem is the SORTLIST.BAS program:

```
' SORTLIST.BAS
' This program gets names and addresses from the user, sorts
'   them alphabetically by last name, and stores them in the
'   sequential file NAMES.TXT.

DECLARE SUB AddNamesToFile ()       ' declare procedures
DECLARE FUNCTION NumberOfNamesInFile% ()
DECLARE SUB CopyFileToArrays (names$(), addresses$(), numOfItems%)
DECLARE SUB ShellSort (names$(), addresses$(), numOfElements%)
DECLARE SUB CopyArraysToFile (names$(), addresses$(), numOfItems%)
DECLARE SUB DisplayNewFile ()

OPTION BASE 1                       ' set base of arrays to 1
```

(continued)

continued

```
CLS                                 ' clear screen

AddNamesToFile                      ' call sub to get input from user
numOfNames% = NumberOfNamesInFile%  ' call function to get num of names

DIM names$(numOfNames%)             ' dimension array to hold names
DIM addresses$(numOfNames%)         ' dimension array to hold addresses

' copy names and addresses in file to arrays in preparation for sorting
CopyFileToArrays names$(), addresses$(), numOfNames%

' sort names$ and addresses$ arrays alphabetically by last name
ShellSort names$(), addresses$(), numOfNames%

' copy sorted arrays back to file
CopyArraysToFile names$(), addresses$(), numOfNames%

' display new file on screen
DisplayNewFile

END

SUB AddNamesToFile

OPEN "NAMES.TXT" FOR APPEND AS #1   ' open for append so we don't lose
                                    '   any existing names and addresses
PRINT "This program adds names and addresses to the file NAMES.TXT"
PRINT "and then sorts the file alphabetically."
PRINT
PRINT "Enter names in Lastname, Firstname format.  Type END to quit."
PRINT

DO WHILE (fullName$ <> "END")       ' get names until user enters END
    LINE INPUT "  Name (Last, First):  "; fullName$
    IF (fullName$ <> "END") THEN    ' use LINE INPUT to allow commas
        PRINT #1, fullName$         ' write data to file
        LINE INPUT "  Address:  "; address$
        PRINT #1, address$
```

(continued)

continued

```
    END IF
    PRINT
LOOP

CLOSE #1

END SUB

SUB CopyArraysToFile (names$(), addresses$(), numOfItems%)

OPEN "NAMES.TXT" FOR OUTPUT AS #1   ' open for output to overwrite
                                    '   out-of-order entries
FOR i% = 1 TO numOfItems%
    PRINT #1, names$(i%)            ' write array contents to file
    PRINT #1, addresses$(i%)
NEXT i%

CLOSE #1

END SUB

SUB CopyFileToArrays (names$(), addresses$(), numOfItems%)

OPEN "NAMES.TXT" FOR INPUT AS #1   ' open for input to get file data

FOR i% = 1 TO numOfItems%
    LINE INPUT #1, names$(i%)      ' read file contents
    LINE INPUT #1, addresses$(i%)
NEXT i%

CLOSE #1

END SUB
SUB DisplayNewFile

INPUT "Press Enter to view NAMES.TXT.  ", dummy$
PRINT
```

(continued)

continued

```
OPEN "NAMES.TXT" FOR INPUT AS #1    ' open for input to get file data

DO WHILE (NOT EOF(1))               ' until end of file is reached,
    LINE INPUT #1, fullName$              ' read file data
    LINE INPUT #1, address$
    PRINT fullName$; " -- "; address$     ' and display on screen
LOOP

CLOSE #1

END SUB

FUNCTION NumberOfNamesInFile%

OPEN "NAMES.TXT" FOR INPUT AS #1   ' open for input to read file data

count% = 0                         ' counter tracks number of name-and-
                                   '   address items
DO WHILE (NOT EOF(1))              ' until end of file is reached,
    LINE INPUT #1, fullName$       '   read name and address
    LINE INPUT #1, address$
    count% = count% + 1            ' increment item counter
LOOP

CLOSE #1

NumberOfNamesInFile% = count%      ' return item count to main program

END FUNCTION

SUB ShellSort (names$(), addresses$(), numOfElements%)
' You can find a discussion of the Shell Sort in Chapter 9.
' Note that this version sorts two arrays based on the contents
'   of names$().

span% = numOfElements% \ 2
```

(continued)

continued

```
DO WHILE (span% > 0)
    FOR i% = span% TO numOfElements% - 1
        j% = i% - span% + 1
        FOR j% = (i% - span% + 1) TO 1 STEP -span%
            IF names$(j%) <= names$(j% + span%) THEN EXIT FOR

            ' swap array elements that are out of order
            SWAP names$(j%), names$(j% + span%)
            SWAP addresses$(j%), addresses$(j% + span%)
        NEXT j%
    NEXT i%

    span% = span% \ 2
LOOP

END SUB
```

CHAPTER 11

1. The two components are the video adapter and the monitor. The video adapter determines the screen modes, generates the text and graphics to be displayed, and generates the colors. The monitor simply displays whatever the video adapter sends it.
2. In text mode, the video adapter can display only alphanumeric characters. In graphics mode, the video adapter can display both text and graphics shapes (pixels, lines, boxes, circles, etc.).
3. LOCATE lets you position the text cursor on the output screen.
4. One possible solution to this problem is the MOVENAME.BAS program.

```
' MOVENAME.BAS
' This program "moves" a name across the screen.

CONST DELAY% = 400

CLS

INPUT "Please enter your first name:  ", name$
LOCATE 10, 1
PRINT name$

FOR i% = 2 TO 70
    LOCATE 10, i% - 1
    PRINT SPACE$(10)
    LOCATE 10, i%
    PRINT name$

    FOR j% = 1 TO DELAY%        ' delay loop
    NEXT j%

NEXT i%
```

5. SCREEN changes your video adapter's mode and determines the screen resolution and (if applicable) the number of colors your video adapter can display.
6. The PSET and PRESET statements set individual pixels on the screen at specified locations. The PSET statement sets the pixel in the current foreground color; the PRESET statement sets the pixel in the current background color.
7. Absolute coordinates are calculated using a starting point (*0*, *0*), which is the upper-left corner of the screen. Relative coordinates use the last plotted point as the starting point for calculation.
8. True, provided you use the B and F options.
9. False.
10. One possible solution to this problem is the CIRCMOVE.BAS program.

```
' CIRCMOVE.BAS
' This program "moves" a circle across the screen.

CONST DELAY% = 50

SCREEN 1                    ' change this value as needed for your adapter

CIRCLE (20, 100), 20

FOR i% = 21 TO 299      ' assumes 320 x 200 resolution
    CIRCLE (i% - 1, 100), 20, 0      ' erase previous circle
    CIRCLE (i%, 100), 20             ' draw new circle

    FOR j% = 1 TO DELAY%             ' delay loop
    NEXT j%

NEXT i%
```

11. One possible solution to this problem is the ANTHEM.BAS program:

```
' ANTHEM.BAS
' This program plays the opening bars of "The Star-Spangled Banner."

CLS

INPUT "Press Enter to begin...", dummy$

FOR i% = 1 TO 12
    READ note%, duration%
    SOUND note%, duration%
NEXT i%

DATA 349, 4, 294, 4, 233, 8, 294, 8, 349, 8, 466, 16
DATA 587, 4, 523, 4, 466, 8, 294, 8, 330, 8, 349, 16
```

CHAPTER 12

1. a. Run your program.

   b. Observe errors and trouble spots in program execution.

   c. Using printouts, programming tools, and your knowledge of QBasic syntax, study the statements that produced the error.

   d. Fix the error and test the program.

2. A syntax error is a programming mistake that violates the rules of QBasic. A run-time error is an error that causes a program to stop unexpectedly during execution.

3. No one correct answer here. Our favorites are the Step and Toggle Breakpoint commands.

4. The F7 function key executes the current line and the lines between the current line and the cursor at full speed.

5. Set Next Statement sets the line the cursor is on as the next statement to be executed.

6. Again, no one correct answer here (especially with so many to choose from!). Our picks for the worst are logic errors resulting from incorrect arithmetic or faulty conditional expressions.

7. DIM and NEXT are misspelled. Here is the correct version (GOODBEAR.BAS):

```
' GOODBEAR.BAS
' This program is the corrected version of BADBEAR.BAS.

CLS

DIM bears$(5)                             ' dimension string array

PRINT "Enter the names of your five favorite bears."
PRINT

FOR i% = 1 TO 5                           ' get 5 strings
    INPUT "Bear:  ", bears$(i%)
NEXT i%
```

(continued)

continued

```
PRINT
PRINT "You entered the following bears:"
PRINT

FOR i% = 1 TO 5                          ' print 5 strings
     PRINT bears$(i%)
NEXT i%
```

8. The INPUT statement should be a LINE INPUT statement to handle the comma in the user's input, and the conditional expression in the IF statement should use a greater-than operator (>) instead of a less-than operator (<). Here is the correct version (GOODNAME.BAS):

```
' GOODNAME.BAS
' This program separates first and last names and prints them.

CLS

PRINT "Enter your first and last name in the following format:  ";
PRINT "Lastname, Firstname"
PRINT

LINE INPUT "Name:  ", fullName$

commaLocation% = INSTR(1, fullName$, ",")

IF (commaLocation% > 0) THEN
    lastName$ = LEFT$(fullName$, commaLocation% - 1)
    firstName$ = RIGHT$(fullName$, LEN(fullName$) - commaLocation% - 1)

    PRINT
    PRINT "What a lovely name!  It's so nice to meet you, ";
    PRINT firstName$; " "; lastName$; "!"
ELSE
    PRINT
    PRINT "Name not in Lastname, Firstname format."
END IF
```

Index

References to figures and illustrations are in italics.

Special Characters
! (single-precision type suffix) 50–51, *53*, 56, 196
"" (string delimiters) 233–34
(double-precision type suffix) 50–51, 56, 196
(template character) 200
$ (string type suffix) 50, 51, 196, 236
$ (template character) 200
$ (wildcard character) 305, 311
% (integer type suffix) 50–51, 53, 196
& (long integer type suffix) 50–51, 54
' (comment indicator) 63
() (precedence operators) 69–71
* (multiplication operator) 62
* (wildcard character) 305, 311
+ (addition operator) 62
+ (concatenation operator) 242
, (comma separator)
  file output 285, 289
  input 258–60
  in numbers 54
  screen output 42–44
– (subtraction operator) 62
. (template character) 200
/ (division operator) 62, 67
; (semicolon separator)
  file output 285
  input 259
  screen output 42–44
< (relational operator) 82, 272
<= (relational operator) 82, 272, 273
<> (relational operator) 82, 272
= (assignment operator) 242
= (relational operator) 82, 272
> (relational operator) 82, 272
>= (relational operator) 82, 272
? (question mark) 61
\ (integer-division operator) 62, 65–67
\\ (template characters) 200
^ (exponentiation operator) 62, 65–67

A
ABS function 72
absolute coordinates 358
active window
  changing 179
  defined 14
  Immediate window as 14–15
adapters, video. *See* video adapters
ADDTHEM2.BAS program 192
ADDTHEM.BAS program 190
algorithms 5
AND logical operator 86–87
animation
  FALLING.BAS program 345–46
  LOCATE-2.BAS program 343
  with LOCATE statement 342–46
APPEND mode (OPEN statement) 283, 299
Apple Macintosh, QuickBasic for 406
arcs 370–73
ARGUMENT.BAS program 173–75
argument list 172
arguments and parameters for procedures
  modifying 173
  passing, to a subprogram 173–75
  relationship of 172–73
arguments for statements and functions
  defined 39
  functions as 41
  multiple, using separators 42–44
  numeric expressions 40–41
  text and strings 48
arrays
  bounds 225–28
  data types 194–97, 223–24
  dimensioning 196–98, 216–20
  dynamic 206–8
  elements 198–203
  errors 220–25, 390
  fixed-length strings 239
  index 198
  index vs. value 222–23
  searching 208–14
  setting lowest subscript 198
  static 206
  storing values in 199–201, 203–5
  two-dimensional 214–20
  usage 95–96, 201–3
  usage, with procedures 215
arrow keys 15
ASC function 271–72
ASCII.BAS program 270
ASCII character set 268–69, 358, 424 (table). *See also* control characters; IBM extended character set
ATN function 72
AVERAGE.BAS program 177–78

B

Backspace key 26
base, array 198
.BAS extension 30
BASICA language 7
Basic language. *See also* QBasic Interpreter; QuickBasic Compiler
  books 405
  converting programs to QBasic 431–40
  described 6
  instructions (*see* instructions, Basic)
  origin 7
  versions 406
BIKEINFO.BAS program 201–3
Boolean expressions. *See also* conditional statements; relational operators
  creating 82–83
  numeric expressions vs. 84
Borland, Russell 405
BOUNDS.BAS program 226–28
boxes 364–66. *See also* IBM extended character set
breakpoints 394
BUDGET.BAS program 141–43
BUGSHAKE.BAS program 386–89

C

CARDATA.BAS program 286–87
case of strings 50, 244–46
CGA (Color Graphics Adapter) 333
characters. *See also* strings; text
  ASCII (*see* ASCII character set; IBM extended character set)
  control (*see* control characters)
  converting, to/from ASCII code 270–71
  printing repeated 260–61
  reading, from keyboard 59–62, 358
CHDIR command (MS-DOS) 304
CHDIR statement 304
check boxes 412
Chorus subprogram 158–59, 165
CHR$ function 270–71
CIRCLE-1.BAS program 369–70
CIRCLE-2.BAS program 372–73
circles 369–70
CIRCLE statement
  arcs 370–73
  circles 369–70
Clear All Breakpoints command (Debug menu) 385
Clipboard 22, 23
CLOSE statement 282, 284–85
CLS statement 19–20, 302, 335
COINS.BAS program 290–98
Color Graphics Adapter (CGA) 333
colors
  filling boxes with 365
  foreground and background 122–25, 356
  QBasic environment 420–21
COLOR statement
  errors 355
  syntax 123, 356
  usage 122–25
column position number 13
command buttons 412
COMMAND command (MS-DOS) 304
commands. *See* menu commands
comma separator
  file output 285, 289
  input 258–60
  screen output 42–44
COMMON SHARED statement 169, 215
COMPARE.BAS program 271–72
compilers, Basic 406, 407
computers
  as heroes or villains 4
  programming (*see* programming)
  reasons for using 2–3
concatenation 241–43
conditional statements. *See also* looping statements
  Boolean expressions 82–83
  IF...THEN...END IF statement 84–97
  introduction 80–*81*
  SELECT CASE...END SELECT statement 98–113
  selecting 113
constants
  declaring 75–76, 234
  string 233–35
  in subprograms 158
CONST statement 75–76, 234
Continue command (Run menu) 384
control characters 425–26 (table). *See also* ASCII character set; IBM extended character set
control keys 14
coordinates, screen
  absolute vs. relative 358
  graphics mode 351
  text mode *334*
Copy command (Edit menu) 21–27
COS function 72
Craig, John Clark 405
Ctrl-Y, deleting lines with 26–27
cursor
  blinking 14
  defined 12, 13
  insert 27–28
  overstrike 28–29
  text 335
Cut command (Edit menu) 21–27

D

DAIRY.BAS program 299–302
data
  arrays 194–98
  in DATA statements 186–94
  file input 293–303
  file output 285–92
  keyboard input 141–43
database program 313–28
DATAEND.BAS program 204–5
data pointer 191
DATA statement
  data pointer 191
  data-type matches 188–89
  end-of-data marker 189
  rereading values 192–94
  usage 186–89
data types
  arrays 194–95, 196–97, 223–24
  errors 390
  matching, in READ and DATA statements 188–89
debugging. *See also* errors
  avoiding bugs 398–99
  breakpoints 394
  common programming errors 390
  demonstration 391–98
  menu commands 383–86
  overview 382
  single-stepping 394–95
  tracking variables with PRINT 386–89
  types of errors 383
Debug menu 384–85
decision making. *See* conditional statements
DECLARE statement 161
DEFSTR statement 236
degrees, converting, to radians 371
DEL command (MS-DOS) 304
Del key 26
DESSERT.BAS program
  debugging 394–98
  running 391–93
dialog boxes
  elements *411*–12
  keyboard use 412–13
  mouse use 412
  scroll bars 413
DICE.BAS program 149
dimensioning arrays 196–98, 216–20
DIM statement
  AS STRING clause 236
  dimensioning arrays 196–98, 216–20
  errors 221–22
direction keys 15
directories of files 305–6
DIR/W command (MS-DOS) 304
Display command (Options menu) 420, *421*
displays 332, 333
DO-1.BAS program 135
DO-2.BAS program 136–37
DO-3.BAS program 137–38
DO-4.BAS program 139–40
DO...LOOP statement
  DO UNTIL loops 137–39
  DO WHILE loops 133–37
  EXIT DO statement 226
  nesting loops 139
  usage 141
DOS operating system. *See* MS-DOS operating system
double-precision variables 50, 55, 57
duplicate definition errors 234
dynamic arrays 206–8
DYNAMIC.BAS program 207–8

E

editing
  keyboard commands for 415
  programs 20–29
EGA (Enhanced Graphics Adapter) 333
ELSEIF.BAS program 95–97
end-of-data marker
  arrays 203–5
  READ and DATA statements 191–92
END statement 164
Enhanced Graphics Adapter (EGA) 333
EOF function 282, 295–98
ERASE command (MS-DOS) 304
errors. *See also* debugging
  arrays 220–25
  common 390
  duplicate definition 234
  illegal function call 336, 355
  out-of-range 224–25
  overflow 58
  redo from start 224
  size 55
  subscript out of range 225, 226–28
  types of 383
Exit (File menu) 34
EXIT DO statement 226
EXIT FOR statement 209
EXP function 72
exponential notation 146
exponentiation operator (^) 62, 67

F

FALLING.BAS program 345–46
fields, storing data in 289–92
filenames
  conventions 30
  directories, wildcard characters, and 304–5

files
  closing 284–85
  creating and opening 283–84
  database program with sequential 313–28
  deleting 311–13
  directories of 304–6
  input from 293–303
  MS-DOS commands to manage 306–9
  numbering 283
  output to 285–92
  QBasic equivalents of MS-DOS commands to manage 303–13
  random-access 282
  renaming/moving 309–10
  statements/functions to access 282 (table)
FILES statement 304–6
fixed-length strings 235, 238–41
floating-point data types
  defined 49
  double-precision 55, 57
  single-precision 55, 56
  type-declaration characters 50–51
floppy disk systems, starting QBasic from 10–11
flowchart *5*
FOR-1.BAS program 119–20
FOR-2.BAS program 120–22
FOR-3.BAS program 122–24
FOR-4.BAS program 124–25
FOR-5.BAS program 126–27
FOR-6.BAS program 128–29
FOR-7.BAS program 130–31
formfeed 299
formula. *See* numeric expressions
FOR...NEXT statement
  COLOR statement with 122–25
  creating loops 118–22
  exiting loops with EXIT FOR 209
  nesting loops 129–31
  SOUND statement with 125–27
  STEP clause 128–29
  usage 141
frequencies, sound 374, *375*
FRIENDS.BAS program 293–95
FRUITS.BAS 287–89
FUNCTION...END FUNCTION statement. *See* function procedures
function keys
  F2 159
  F4 385
  F5 384
  F6 14–15
  F8 384
  F9 385
  F10 384
function procedures
  creating 176–78
  defined 157, 175–76
  program layout with *160*
function procedures, *continued*
  subprograms vs. 178
  usage 180
functions
  as arguments 41
  errors 355
  mathematical 71–74
  QBasic 429–30
  statements vs. 38
  string 243–67
  user-defined (*see* function procedures)

G

GetData subprogram 394–96
GETLEFT.BAS program 251–52
GETMID.BAS program 253–55
GETNAMES.BAS program 199–201
GETRIGHT.BAS program 250
GetText subprogram 264, 273
global arrays 215
GLOBAL.BAS program 169–70
global variables
  declaring 169–70
  local variables vs. 168–69
graphics
  absolute coordinates vs. relative coordinates 358
  colors 356 (*see also* colors)
  complex shapes 362–73
  coordinates vs. text coordinates 351
  illegal function calls 355
  LINE statement 362–69
  modes 333–34, 346, 349 (table)
  overview 346–48
  positioning pixels 352–62
  PRESET statement 356–58
  PSET statement 352–54, 359–62
  resolution 347–48
  setting 348–50
  text mode (*see* text mode)
  video equipment 332–33
GREETING.BAS program 257–58
GUESS.BAS program 147–48
GW-BASIC language
  converting programs to QBasic 431–40
  origin 7
  running, programs under QBasic 433–39
  QBasic vs. 433

H

hard disk, starting QBasic from 10
hardware, defined 4
HEADER.BAS program 261
help, on-line
  accessing 418–19
  copying programs from 419
  printing portions of 420

Hercules Graphics Card (HGC) 333
Hergert, Douglas 405
HGC (Hercules Graphics Card) 333
HUNT.BAS program 262–63
hyperlinks 418

I

IBM extended character set 269–70, 425 (table). *See also* ASCII character set; control characters
IBM PC video adapters. *See* video adapters
IF-1.BAS program 85
IF-2.BAS program 87
IF-3.BAS program 88
IF-4.BAS program 90
IF-5.BAS program 92–93
IF...THEN...END IF statement
  ELSE clause 89–91
  ELSEIF clause 94–97
  END IF clause 91–93
  logical operators 86–89
  usage 84–86
illegal function call errors 336, 355
immediate window
  as active window 14–15
  testing program lines 65
index, array
  array value vs. 222–23
  defined 198
information. *See* data
INKEY$ function 358
input, file. *See* files
input, keyboard. *See* keyboard
input, mouse. *See* mouse
INPUT# statement 282, 293–95
INPUT mode (OPEN statement) 283
INPUT statement
  LINE INPUT vs. 258–60
  reading characters with 59–62
insert cursor 27–28
INSTR function 262–67
instructions, Basic. *See also* functions; *names of individual functions and statements;* statements
  order of 19–20
  statements vs. functions 38
  syntax 38–44
integer data type
  defined 49
  long 52, 54–55
  regular 52, 53–54
  subscripts 221
  type-declaration characters 50–51
integer division 62, 65–67
INT function 147–48

J

Jamsa, Kris, 405

K

Kemeny, John G. 7
keyboard
  accessing IBM extended character set from 269
  dialog boxes usage 412, 413 (table)
  editing commands 415
  input 59–62
  menu-command equivalents 414
  scroll bars usage 413
  selecting menus and commands 411
  selecting text 417
keywords, Basic 6, 18, 50
KILL.BAS program 311–13
KILL statement 304, 311–12
Kurtz, Thomas E. 7

L

Lammers, Susan 405
LBOUND function 225–28
LCASE$ function 245–46
LEFT$ function 249, *251*, 252
LEN function 246–48
LINE-1.BAS program 363–64
LINE-2.BAS program 365–66
LINE INPUT# statement 282, 298–301
LINE INPUT statement 258–60, 299
lines 362–64. *See also* IBM extended character set
LINE statement
  boxes 364–66
  complex shapes 366–69
  simple lines 362–64
  relative coordinates 367–68
list boxes 412
literal string constants 233–35
local arrays 215
LOCAL.BAS program 170–71
local variables
  declaring 170–71
  global variables vs. 168–69
LOCATE-1.BAS program 337–42
LOCATE-2.BAS program 343
LOCATE-3.BAS program 343–44
LOCATE statement
  animation 342–46
  errors 355
  syntax 336
  usage 337–42
logical operators
  AND 86–87
  NOT 89
  OR 87–89
logic errors 383, 390
long integer data type 50, 52, 54–55. *See also* integer data type
looping statements. *See also* conditional statements
  arrays with 221–22

looping statements, *continued*
  DO...LOOP statement 133–39
  endless 390
  exiting 209
  FOR...NEXT statement 118–31
  introduction 118
  nesting 129–31, 133, 140–41
  selecting 141
  usage 141–50
  WHILE...WEND statement 131–33
LOWER.BAS program 246
LPRINT statement 299
LTRIM$ function 256–58, 339

M

main program
  layout showing subprograms, functions, and *160*
  usage 180–81
mathematical functions 71–74
mathematical operators
  mathematical functions 71–74
  numeric expressions 67–71
  QBasic 62 (table)
  special purpose 65–67
  usage 64–65
MAXSALES.BAS program 212–14
MCGA (Multicolor Gate Array) 333
MDA (Monochrome Display Adapter) 333
memory for arrays 196–98
menu bar 14
MENU.BAS program 247–48
menu commands. *See also names of commands*
  debugging 383–86
  keyboard equivalents 414
  selecting 410–11
  selecting text 416–17
menu name 14
menus and options, QBasic 409–22
Microsoft Word for Windows 2.0 Macros (Borland) 405
MID$ function 252–56
MKDIR command (MS-DOS) 304
MKDIR statement 304
modes
  graphics 333–34, 346, 349 (table)
  OPEN statement 283
  text (*see* text mode)
MOD operator 62, 65–67
monitors 332, 333
Monochrome Display Adapter (MDA) 333
mouse
  dialog boxes usage 412
  scroll bars usage 413
  selecting menus and commands 410
mouse, *continued*
  selecting text 416–17
  switching between windows 14
mouse pointer 13
MS-DOS operating system
  commands, exiting QBasic to run 306–9
  commands, QBasic equivalents of 303–13
  version 5 of 8
MS-DOS QBasic (Jamsa) 405
Multicolor Gate Array (MCGA) 333
musical notation 377–79
MUSICDB.BAS program
  creating 319–28
  running 313–19

N

NAME statement 304, 309–10
NESTING.BAS program 140
nesting loops 129–31, 133, 139–40
New command (File menu) 32–33
New Function command (Edit menu) 176
NEWNAME.BAS program 309–10
NEWS.BAS program 242–43
NEWSONG.BAS program 167–68
New SUB command (Edit menu) 159, 165
NOT logical operator 89
numbers
  Basic's view of 52
  commas in 54
  converting, to strings 339
  printing 54
numeric expressions
  as arguments 40–41
  conditional expressions vs. 83
  defined 67–68
  mathematical functions 71–74
  multiple operators 68
  order of calculation (precedence) 68–69
  precedence operators (parentheses) 69–71
numeric variables
  changing value of 59
  defined 48–49
  double-precision 57
  floating-point 49, 52, 55–57
  integer 49, 52–55
  long integer 52, 54–55
  selecting type 57–58
  single-precision 56
  size errors 55
  type-declaration characters 50–51

O

Open command (File menu) 33–34
opening and running existing programs 33–34
OPEN statement 282, 283–84

operators. *See* logical operators; mathematical operators; relational operators
OPTION BASE statement 198
option buttons 412
options and menus, QBasic 409–22
Options menu, debugging command on 385–86
OR logical operator 87–89
out-of-range errors 224–25
output, file. *See* files
output, printer. *See* printing
output, screen. *See* screen
OUTPUT mode (OPEN statement) 283
Output Screen command (View menu) 385
overflow errors 58
overstrike cursor 28–29

P

parameter list 161, 172
parameters. *See* arguments and parameters for procedures
passing arguments to subprograms 172–75
Paste command (Edit menu) 21–27
PHONE.BAS program 237–38
PHONEFIX.BAS program 240–41
PIG.BAS program 235
pixels
  defined 348
  positioning, with coordinates 358–62
  positioning, with PSET and PRESET statements 352–58
PRESET.BAS program 357–58
PRESET statement 356–62
PRINT# statement 282, 285–87
PRINT# USING statement 282, 287–89
Print command (File menu) 419–20
printing
  LPRINT statement 299
  portions of help system 420
  programs 419
PRINT statement
  without arguments 40
  concatenation 242
  debugging with 386–89
  LCASE$ function and 246
  LEN function with 247–48
  multiple arguments 42–44
  separators 44
  syntax 39–41
  text viewports 302
  UCASE$ function and 244–45
  usage 17–18
PRINT USING statement 200
procedures
  advantages 154–57
  function procedures (*see* function procedures)
  subprogram procedures (*see* subprogram procedures)
Procedure Step command (Debug menu) 384
Programmers at Work (Lammers) 405
programming
  algorithms 5
  common errors 390
  introduction 2–8
  modular (*see* function procedures; subprogram procedures)
  in QBasic environment 17–34
programs
  changing saved 31–32
  comments 63
  converting, to QBasic 431–40
  creating 17–18
  debugging (*see* debugging)
  defined 4
  editing 19–29
  errors (*see* errors)
  layout showing main program, subprograms, and functions *160*
  naming 30
  opening and running existing 33–34
  running 18–19
  saving 29–30
  starting new 32–33
  usage of main programs and procedures 180–81
PSET.BAS program 353–54
PSET statement 352–56, 358–62

Q

QBasic interpreter. *See also* Basic language; QuickBasic Compiler
  advantages 439–40
  books 405
  converting GW-BASIC programs to 431–40
  debugging (*see* debugging)
  described *xi–xiv*, 6
  environment 11–16
  equivalents of MS-DOS commands 303–13
  GW-BASIC vs. 433
  menus and options 409–22
  origins 7
  programs (*see* programs)
  quitting 34
  running GW-BASIC programs under 433–39
  starting 10–11
  start-up options 422
QuickBasic Compiler 406, 407
quitting QBasic 34

R

radians, converting degrees to 371
random-access files 282
random numbers 144–50
READ statement
  assigning values to variables 190

READ statement, *continued*
  data pointer 191
  data-type matches 188–89
  end-of-data marker 191–92
  rereading data 192–94
  usage 186–89
records. *See* files
redo from start errors 224
reference, QBasic 409–22
reference bar 14–15
regular integer data type. *See* integer data type
relational operators
  with Boolean expressions 82 (table), 83
  with strings 272 (table), 273
relative coordinates 358–62
REM statement 63
RENAME command (MS-DOS) 304
REPEAT.BAS program 264–67
resolution, low vs. high graphics 347–48
Restart command (Run menu) 384
RESTORE statement 192–94
RIGHT$ function 249, *250*
RMDIR command (MS-DOS) 304
RMDIR statement 304
RND-1.BAS program 144–45
RND-2.BAS program 145–46
RND function 144–46
row position number 13
RTRIM$ function 256–58
RUNDOS.BAS program 307–9
Run menu, debugging commands on 384
Running MS-DOS (Wolverton) 8, 406
run-time errors 383

S

SALES.BAS program 217–20
Save As command (File menu) 29
saving programs 29–30, 32
screen
  clearing 19–20, 302, 305
  coordinates in graphics mode 351, 358
  coordinates in text mode *334*
  errors 390
  output 19, 200, 301–3
  QBasic environment 11–16
  text viewports 301–3
SCREEN statement
  errors 355
  modes 348–51
scroll bars 413
SEARCH.BAS program 209–11
SELECT-1.BAS program 100
SELECT-2.BAS program 102–3
SELECT-3.BAS program 106
SELECT CASE...END SELECT statement
  CASE ELSE clause 102–4
  CASE IS clause 104–6
SELECT CASE...END SELECT statement, *continued*
  multiple conditions 111–13
  TO keyword 106–11
  usage 98–102, 113
semicolon separator
  file output 285
  input 259
  screen output 43–44
separators in multiple arguments 42–44
sequential files. *See* files
Set Next Statement command (Debug menu) 385
SGN function 72
shapes, complex
  arcs 370–73
  boxes 364–66
  circles 369–70
  coordinates 366–69
  lines 362–64
Shell, Donald 274
Shell Sort algorithm 274
ShellSort subprogram 273, 274
SHELL statement 304, 306–9
SHOW.BAS program 305–6
simulation program 149–50
SIN function 72
single-precision data type 50, 55, 56
software, defined 4. *See also* programs
SONG2.BAS program 161–68
SONG.BAS program 154–56
sorting strings 273–77
SOUND-1.BAS program 376–77
SOUND-2.BAS program 378–79
sound programming
  animation and 344–46
  frequencies 374, 375 (table)
  musical notation 377–79
  playing scales 376–77
  playing songs 378–79
  sound-effects generation 125–27
SOUND statement 126, 374–79
SPACE$ function 260–61
Split command (View menu) 179
SQR function 72–74
Start command (Run menu) 18, 384
statements. *See also names of statements*
  functions vs. 38
  QBasic 428–29
STEP-1.BAS program 360–62
STEP-2.BAS program 368–69
Step command (Debug menu) 384
STR$ function 339
STRING$ function 260–61
string constants
  literal 233–34
  symbolic 234
  usage 235

strings. *See also* characters; text
  arrays 196–98
  case of 244–46
  changing value of 59
  comparing 268–77
  concatenation 241–43
  constants 233–35
  defined 40, 48
  determining length of 246–48
  fixed-length 235, 238–41
  formatting, for output 200
  functions 243–67
  input 258–60
  overview 232–33
  retrieving substrings 249–67
  sorting 273–77
  trimming 256–58
  type-declaration 50–52, 196, 236
  variable-length 235, 236–38
STRSORT.BAS program 273–77
SUB...END SUB statement. *See* subprogram procedures
subprogram procedures 157–68
  arrays with 215
  calling 161–68
  creating 157–61
  declaring 161
  function procedures vs. 178
  passing arguments to 172–75
  program layout including 160
  syntax 158–59
  usage 180
  variables with 168–75
SUBs command (View menu) 159, 166–67, 176
subscript out of range errors 225, 226–28
subscripts 198, 221
SWAP statement 274
symbolic string constants 234
syntax 6, 38–44
Syntax Checking command (Options menu) 386
syntax errors 383

T

tabular form 200
TAN function 72
template characters 200
TEST-VID.BAS program 349–51
text. *See also* characters; strings
  changing 27–29
  deleting 26–27
  input 298–301
  selecting 416–17
text boxes 412
text mode
  animation 342–46
  cursor 335
  graphics mode vs. 333–34
text mode, *continued*
  LOCATE statement 336–46
  screen coordinates *334*
text viewports 301–3
TO-1.BAS program 109
TO-2.BAS program 110–11
TO-3.BAS program 112–13
Toggle Breakpoint command (Debug menu) 385
Trace On command (Debug menu) 385
TVHOURS.BAS program 193–94
two-dimensional arrays
  declaring 216–20
  defined 214–16
type-declaration characters 50–51

U

UBOUND function 225–28
UCASE$ function 244–45
UPPER.BAS program 244–45

V

variable-length strings 225, 236–38
variables
  arrays (*see* arrays)
  case of 50
  errors in naming 390
  global vs. local 168–71
  naming 49–50
  overview of 49–51
  passing, as arguments to subprograms 172–75
  type-declaration 49, 50–51
  types 48–49
  usage 51, 59–62, 64
VGA (Video Graphics Array) 333
video adapters
  described 332–33 (table)
  identifying 349–51
  resolution 347–48
Video Graphics Array (VGA) 333
VIEW.BAS program 302–4
VIEW menu, debugging command on 385
viewports, text. *See* text viewports
VIEW PRINT statement 301–3
view window 14, 16

W

Waite Group's Microsoft QuickBASIC Primer Plus, The 405
Waite Group's MS-DOS QBasic Programmer's Reference, The 405
WHILE.BAS program 132
WHILE...WEND statement
  creating loops 131–33
  usage 141
wildcard characters 305, 311
window key (F6) 14–15

windows. *See* active window; immediate
    window; view window
Wolverton, Van 8, 406
WRITE# statement 282, 285, 289–92

Z

zoom control 15–16

Michael Halvorson

Michael Halvorson received his B.A. in computer science from Pacific Lutheran University in 1985 and has been employed as a programmer, technical editor, and community college instructor. He is currently an acquisitions editor at Microsoft Press.

David Rygmyr

David Rygmyr got started in computers in 1977 by preparing daily feedings of paper tapes and punch cards for a room-size UNIVAC computer. He has since extended his expertise into computer science, electronics, and digital control/robotics. He is currently technical manager at Microsoft Press, which he joined in 1984.

Halvorson and Rygmyr are the authors of *Learn BASIC Now* and *Learn BASIC for the Apple Macintosh Now,* published by Microsoft Press.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Principal proofreader: Shawn Peck
Principal typographer: Carolyn Magruder
Interior text designer: Darcie S. Furlan
Principal illustrators: Rebecca Geisler-Johnson and Kim Eggleston
Cover designer: Thomas A. Draper
Cover color separator: Rainier Color

Text composition by Microsoft Press in Times Roman with display type in Times Bold, using the Magna composition system and the Linotronic 300 laser imagesetter.

Printed on recycled paper stock.

ORDER CARD

For information on the Companion Disk for RUNNING MS-DOS QBASIC and ordering information outside of the U.S., please see page x.

YES... please send me ________ copies of the Companion Disk for RUNNING MS-DOS QBASIC at $18.95 each (U.S. funds only) . . . . $ ________

Sales Tax: Add the applicable sales tax in the following states: AZ, CA, CO, CT, DC, FL, GA, HI, ID, IL, IN, IA, KS, KY, LA, ME, MD, MA, MI, MN, MS, MO, NE, NV, NJ, NM, NY, NC, ND, OH, OK, PA, RI, SC, TN, TX, UT, VA, VT, WA, WV, WI, AND WY. . . . . $ ________

Shipping Charges: $5.00 per disk set (domestic orders)
$15.00 per disk set (foreign orders) . . . . $ ________

Please check correct box: ☐ 5.25-inch format 097-000-502 ☐ 3.5-inch format 097-000-505

TOTAL $ ________

NAME ________________________________

ADDRESS ________________________________

________________________________ (   ) ________________
DAYTIME PHONE NUMBER

CITY ________________ STATE ________ ZIP ________

Payment: ☐ Check/Money Order ☐ VISA (13 or 16 numbers) ☐ MasterCard (16 numbers) ☐ American Express (15 numbers)

Credit Card No. ^{1}[ ] ^{2}[ ] ^{3}[ ] ^{4}[ ] ^{5}[ ] ^{6}[ ] ^{7}[ ] ^{8}[ ] ^{9}[ ] ^{10}[ ] ^{11}[ ] ^{12}[ ] ^{13}[ ] ^{14}[ ] ^{15}[ ] ^{16}[ ] ________
EXP. DATE

CARDHOLDER SIGNATURE ________________________________

Please allow 2-3 weeks for delivery.

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 53 BOTHELL, WA

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT PRESS
ATTN: RUNNING MS-DOS QBASIC
COMPANION DISK OFFER
PO BOX 3011
BOTHELL WA 98041-9908

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

The manuscript for th
Press in electronic fo
Microsoft Word.

Principal proofreader:
Principal typographer:
Interior text designer:
Principal illustrators:
Cover designer: Thom
Cover color separator:

Text composition by M
Times Bold, using the
laser imagesetter.

Printed on recycled paper stock.

Microsoft®

RUNNING MS-DOS® QBasic™

MS-DOS QBasic is the exciting, full-featured, modern programming language that comes with MS-DOS versions 5 and 6. The friendly user interface in MS-DOS QBasic and its superior online help, editing, and debugging tools make learning to program easy and enjoyable. If your computer runs MS-DOS 5 or 6, you'll find this book an unbeatable way to learn to use the built-in programming language you get with MS-DOS.

Even if you have little or no programming experience, RUNNING MS-DOS QBASIC is your best and easiest route to understanding programming concepts and writing your first Basic programs. This complete self-study course includes:

A Solid, Hands-On Introduction and Tutorial. You'll find clear information on program design concepts, step-by-step instructions for writing Basic programs, debugging advice, and a handy guide to converting GW-BASIC® programs to MS-DOS QBasic.

Sample Programs. Dozens of programs illustrate good programming technique. They're useful to boot, *and* fun!

Questions and Exercises. Specially designed review questions, answers, and exercises reinforce your proficiency with MS-DOS QBasic.

Note: This book is for owners of the MS-DOS operating system version 5 or 6, which includes the MS-DOS QBasic programming language. If you have an earlier version of MS-DOS, you can learn to program in Basic with *Learn BASIC Now,* a book-and-software package that combines the instruction from RUNNING MS-DOS QBASIC with an on-disk Microsoft QuickBasic™ Interpreter. *Learn BASIC Now* is available at bookstores everywhere, and you can also order it directly from Microsoft Press. If you already own the *Learn BASIC Now* package, you can apply all its programming instruction to MS-DOS QBasic.

U.S.A.	**$22.95**
U.K.	£19.95
Canada	$29.95

[*Recommended*]

Microsoft® PRESS

The Authorized Editions

ISBN 1-55615-340-6

90000

9 781556 153402